Countering Kernel Malware in Virtual Execution Environments

A Thesis
Presented to
The Academic Faculty

by

Chaoting Xuan

In Partial Fulfillment
Of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
December 2009

Countering Kernel Malware in Virtual Execution Environments

Approved by:

Dr. John Copeland
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. George Riley
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Doug Blough
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Alessandro Orso
School of College of Computing
Georgia Institute of Technology

Dr. Raheem Beyah
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: Nov 5, 2009

For my family

ACKNOWLEDGEMENTS

First of all, I wish to express my sincere thanks to my advisor, Dr. John Copeland for his insight guidance, endless patience and support. Without his help, this research could not have been completed. I am indebted to him.

I am also grateful to my committee members, Dr. Doug Blough, Dr. George Riley, Dr. Alessandro (Alex) Orso and Dr Raheem Beyah for their valuable times, professional suggestions and gracious services on my committee.

I would also like to thank people in the Communications Systems Center, Raheem Beyah, Yusun Chang, Chris Lee, Bongkyoung Kwon, Selcuk Uluagac and Myounghwan Lee for their valuable time and friendship.

My most special thanks go to my parents Jiarang Xuan, Xiuying Zhu, and my wife Hu Tu for their ever-lasting encouragement, support and love.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

We present a rootkit prevention system, namely DARK that tracks suspicious Linux loadable kernel modules (LKM) at a granular level by using on-demand emulation, a technique that dynamically switches a running system between virtualized and emulated execution. Combining the strengths of emulation and virtualization, DARK is able to thoroughly capture the activities of the target module in a guest operating system (OS), while maintaining reasonable run-time performance. To address integrity-violation and confidentiality-violation rootkits, we create a group of security policies that can detect all available Linux rootkits. It is shown that normal guest OS performance is unaffected. The performance is only decreased when rootkits attempt to run, while most rootkits are detected at installation.

Next, we present a sandbox-based malware analysis system called Rkprofiler that dynamically monitors and analyzes the behavior of Windows kernel malware. Kernel malware samples run inside a virtual machine (VM) that is supported and managed by a PC emulator. Rkprofiler provides several capabilities that other malware analysis systems do not have. First, it can detect the execution of malicious kernel code regardless of how the monitored kernel malware is loaded into the kernel and whether it is packed or not. Second, it captures all function calls made by the kernel malware and constructs call graphs from the trace files. Third, a technique called aggressive memory tagging (AMT) is proposed to track the dynamic data objects that the kernel malware visits. Last, Rkprofiler records and reports the hardware access events of kernel malware (e.g., MSR register reads and writes). Our evaluation results show that Rkprofiler can quickly expose the security-sensitive activities of kernel malware and thus reduces the effort exerted in conducting tedious manual malware analysis.

CHAPTER 1

INTRODUCTION

Despite years of research in kernel malware detection, kernel malware (rootkits) remains a significant threat to today's Internet security. In fact, most of the legacy systems today, like Windows XP and many Linux distributions, don't have capabilities for preventing the invasions of kernel malware once hackers acquire root privilege and deploy them. Realizing the seriousness of the issue, people have proposed several intrusion prevention approaches to thwart kernel malware. For example, Microsoft 64-bit Windows Server 2008 only allows signed device drivers to load into its kernel [1]. Unfortunately, this approach does hurt the extensibility and usability of a commodity operating system, and its effectiveness is also questionable [2] [3]. The first objective of this research is to develop a new intrusion prevention approach to defend against kernel malware in virtual execution environments, which is complementary to previous intrusion prevention approaches. Concretely, one security software system based on virtual machine monitor (VMM) and hardware emulator is designed and implemented to collaboratively carry out the prevention task toward any suspicious kernel modules. Fine-grained intrusion prevention is achieved by enforcing a set of access control policies at the OS and hardware object level. In the end, this research demonstrates it is possible to create an effective and high-performance intrusion prevention system for kernel malware in commodity operating systems.

Recently, it has been found that hackers have started applying anti-reverse-engineering techniques to rootkits, e.g., packing kernel code, which impedes the static analysis of kernel malware. Therefore, dynamic analysis of kernel malware in

virtualized execution environments has drawn people's attention. Unfortunately, previous dynamic analysis systems are limited in the sense that they focus on the discovery of specific behaviors of kernel malware such as hooking activities [4] [5] and triggering events [6]. The second objective of this research is to build a novel malware analysis system based on hardware emulator that can automatically capture the extensive behaviors of kernel malware, including the kernel functions and data structures that malware call and visit. We expect this system will be able to help people quickly and accurately identify the security-sensitive activities of kernel malware in the future.

The dissertation is organized as below: chapter 2 provides the background information on kernel malware and virtualization security. Additionally, an overview of related research work is presented. Chapter 3 and chapter 4 respectively present the details of kernel intrusion prevention system DARK and kernel malware analysis system Rkprofiler, including designs, implementations and evaluations. Finally, chapter 5 draws conclusions and points out the avenues of future work.

CHAPTER 2

BACKGROUND

2.1 Kernel Malware

Kernel malware refers to kernel rootkits in this thesis. A rootkit is a program designed to take fundamental control of a computer system, without authorization by the system's owners and legitimate managers [7]. Running as a root user (administrator in Windows), a rootkit can arbitrarily access and modify any system resource on a victim machine. This "almighty power" makes the defense against rootkits become one of the most technical challenges to the security community in recent years.

A rootkit that is installed in user space is referred as *user-land rootkit*. One typical technique employed by user-land rootkit is to tamper with system libraries like glibc.so and ntdll.dll to hide processes or files. Contrarily, a rootkit that is installed in the kernel is called *kernel rootkit*. The kernel rootkits are more difficult to address than user-land rootkits, as the former locates at a lower software layer and can do more damages. This research is concentrated on studying the defense mechanisms against kernel rootkits. For ease of the presentation, the term *rootkit* is used to substitute the term *kernel rootkit* in the rest of this dissertation.

The first known rootkit was written for SunOS in 1994 and it was intended to take control of an unresponsive system as a fault handling utility. However, when Internet came to many peoples lives, it became the carrier of all kinds of malware. Soon, these malware writers found that rootkits could help gain access to systems while avoiding detection; rootkit techniques took off then. Rootkits draw public attention in 2005, when Sony BMG caused a scandal by including rootkit software on music CDs which

altered the Windows OS to allow access to anyone who is aware of the rootkit installation. The discovery of this corporate-sponsored malware made many users previously unfamiliar with rootkits wary.

Today, hackers make use of rootkits to achieve multiple attacking goals. At least five goals are observed in practice: *user-space object hiding (HID), reconnaissance (REC), reentry/backdoor (REE), defense neutralization (NEU),* and *privilege escalation (PE).* HID rootkits conceal hackers' presence on a compromised system by hiding files, processes, drivers, network connections, registry entries and others. Administrators can't run regular system software to recognize these hidden objects. The intention of REC rootkits is all about monitoring what people do on a victim computer. Rootkits can sniff packets, intercept keystrokes, or read e-mails. The typical targets of these rootkits are credit card numbers, bank accounts, passwords and email contacts. REE rootkits allow hackers to turn a victim computer into a zombie computer (normally become one node of a botnet) and use it as a staging ground for further abuse such as denial-of-service attack, relaying chat sessions and email spam distribution. NEU rootkits directly attack security software and bypass their defense. The techniques used by NEU rootkits include disabling the security services, modifying the kernel firewall hooks and abusing signature libraries. Last, PE rootkits alter kernel data structures to allow a non-privileged user to have root privilege and carry out tasks that are reserved for super users. As a note, a rootkit could carry multiple attacking goals. For example, FUTo [8] is referred to as both HID rootkit and PE rootkit.

This research also adopts Joanna Rutkowska's rootkit categorization [9] in the interest of pursing a standardized classification. Her model describes three types of kernel malware. Type I includes malware that modifies things that should otherwise

never be modified (code segments, MSRs, BIOS, etc); Type II includes malware that modifies things that should be modified (global variables, some hooks and other data). Most rootkits being used for real-world attacks belong to type I and II rootkits. Type III rootkits just target special hardware features. For example, Blue pill [10] takes advantage of hardware virtualization extensions (Intel's VT [11] and AMD's Pacifica [12]) to hide itself.

2.2 Virtual Machine and Security

A virtual machine (VM) is a software implementation of a machine that executes programs like a real machine. Depending on the computing layers that virtual machines imitate, they are separated into three categories: application virtual machine, OS virtual machine and hardware virtual machine. An application virtual machine is a software layer between operating system and a process (application), and it provides a platform-independent programming environment that abstract underlying software and hardware resources. Java virtual machine (JVM) and Common language runtime (CLR) of .NET framework are two such examples. An OS virtual machine is a system-level compartment (or zone) that has its own files, processes, user and root accounts. The typical examples are Free BSD jails [13] and OpenVZ [14]. A hardware virtual machine allows the sharing of the underlying physical machine resources between different virtual machines and each VM runs its own operating system (namely guest OS). The software layer providing virtualization is called a virtual machine monitor (VMM) or hypervisor, which can run on either bare hardware or host operating system. VMware [15] and Xen [16] are two popular VMMs. Because application virtual machines and OS virtual machines are not effective in defeating kernel malware, hardware virtual machines are the focus of this research.

Gerald J. Popek and Robert P. Goldberg [78] proposed a set of sufficient conditions for computer architecture to efficiently support system virtualization, which is known as Popek and Goldberg virtualization requirements. In short, a genuine virtual machine should possess three properties:

1. *Equivalence:* a program running under the VMM should exhibit a behavior essentially identical to that demonstrated when running on an equivalent machine directly.
2. *Resource control:* the VMM must be in complete control of the virtualized resources.
3. *Efficientcy:* a statistically dominant fraction of machine instructions must be executed without VMM intervention.

Popek and Goldberg further introduced a classification of instructions in an Instruction Set Architecture (ISA) into two groups: privileged instructions and sensitive instructions. They pointed out that, for any computer, a VMM can be constructed if the set of sensitive instructions for that computer is subset of the set of privileged instructions. IBM System/370 and Motorola MC68020 are two examples of natively virtualizable architectures that meet the Popek and Goldberg virtualization requirements. Unfortunately, John Scott Robin and Cynthia E. Irvine [79] shown that X86 architecture violates the Popek and Goldberg virtualization requirements: IA-32 instruction set contains 17 sensitive and unprivileged instructions.

To overcome this limitation, people developed two approaches to implement VMM on X86 CPUs: paravirtualization and direct execution combined with fast binary translation. With paravirtualization, the VMM builder defines the virtual machine interface by replacing nonvirtualizable portions of the original instruction set with easily virtualized and more efficient equivalents. Thus, operating systems must be

ported to run in a virtual machine. For example, Xen [16] (until version 2.0) used to paravirtualization to make virtual machines gain high performance. The second approach runs application programs (in the unprivileged mode) of a VM directly on the CPU and the VMM has a binary translator that controls the kernel code (in the privileged mode) of the VM. The translator translates the privileged code into a similar block, replacing the problematic instructions, which lets the translated block run directly on the CPU. The translation blocks are placed in a trace cache so that translation does not occur on subsequent executions. VMware workstation adopted this approach. On the other hand, CPU vendors also added new hardware features to the X86 architecture to support virtualization. Both Intel and AMD added a new execution mode to the processor that lets a VMM safely and transparently use direct execution for running virtual machines, which is called hardware-assisted virtualization. In addition, the recent releases of Intel and AMD's CPUs also have virtualization supports for memory and IO management, and they improve the virtual machine's performance significantly [80].

Virtualization emerged as a technique for managing mainframes in the 1960s. Virtual machine technology was rapidly gaining acceptance as a fundamental building block in enterprise data centers since the late 1990s. Today, not only are the servers in data center virtualized, but also are corporate desktops being virtualized. We have seen an explosion in growth in development, deployment and implementation of various VM-related products. The main drives that make VM popular are reduced hardware cost, ease of management and simplified maintenance. Although security benefits offered by VM are not a major factor affecting the commercial world in the last decade, it has been predicted that VM security products could take off soon [17].

Virtualization holds unique properties that make it attractive for building novel system security mechanisms, including *isolation, inspection, interposition and replay*. *Isolation* indicates that software running in a VM cannot access or modify the software running in the VMM or in a separate VM. Even if an intruder has completely subverted a VM, he still cannot tamper with security software outside the VM. *Inspection* means that the VMM has access to all the states of a VM: CPU states (e.g. registers), memory, and IO device states. This property makes it difficult for malware in a VM to evade VMM-supported security software, since there is no state in the monitored guest host that the VMM-supported security software cannot see. *Interposition* refers to the capability that the VMM interposes on certain virtual machine operations such as software interrupts and execution of privileged instructions. Intercepting certain VM events can be the key building block in a VMM-based security system. *Replay* is a property that the VMM snapshots a VM, logs nondeterministic VM inputs (like keystrokes) and replays the long-term execution of the VM instructions. This enables VMM to provide arbitrarily detailed observations on what transpired on the compromised VM, even in the presence of non-deterministic attacks and executions.

A number of VMM-based security systems that took advantage of these security-friendly properties have been successfully designed and developed in last six years. For examples, Livewire [18] and VMwatcher [19] are two VMM-based host intrusion detection systems (HIDS) that pull the security software outside of the monitored VM. These systems not only gain greater attack resistance from operating outside the VM, but also benefit from the ability to inspect and interpose the guest OS inside the VM at a hardware level. Terra [20] and Overshadow [21] are another two VMM-based security systems that replies on *isolation, inspection and interposition* to build

trust operating systems and applications. ReVirt [22] and Aftersight [23] apply the *reply* technique of virtual machines to creating fine-grained intrusion analysis systems.

Because operating systems operate at the highest level privilege (also called “ring 0”) in physical machines, kernel malware makes security software on these machines stuck with a difficult chicken-and-egg problem. Once the kernel has been compromised, the detection tools themselves are left unprotected. Thus, the arms race between attackers and defenders rapidly devolves into a complex game of war. One instance is Microsoft PatchGuard [24] and the arms race to disable it is already in full swing [25]. Being named “ring -1”, the emergence of VMM changes the rule of this game: defenders (detection tools in VMMs) have higher system privilege than attackers (kernel malware in VMs) and therefore gain the prior advantage in the game. Unsurprisingly, the studies on VMM-based rootkit defense mechanisms [18] [19] [26] [27] [28] [29] [30] [31] [38] bloom in accordance.

2.3 Hardware Emulation

Hardware emulation is a process that one software program (emulator) duplicates the functions of one computer architecture (CPU and IO devices), so that a virtual machine running over the emulator behaves like (and appears to be) the equivalent running over the real machine. While running on a host CPU architecture, a hardware emulator can simultaneously provide multiple CPU architectures to its VMs. So, you can run Windows Mobile on an ARM CPU, being supported by an emulator that is executed within Red Hat Linux on an X86 CPU. In comparison, a VMM requires that the CPU architecture of a VM is same as the host CPU architecture. Because emulator has to translate every instruction of a VM to the corresponding host instruction and

direct execution is not allowed, emulation imposes higher performance overhead than virtualization. Boch [81] and Qemu [82] are two popular open source hardware emulators. The latter has been extensively used in this research.

The core component of Qemu is the dynamic translator [46]. It performs a runtime conversion of the target CPU instructions to the host instructions. The resulting binary code is stored in a code cache so that it can be reused. The dynamic translator interprets the target code using *decode-dispatch* approach [83]. Qemu uses a simulated Program Counter (PC) of the VM to guide the iteration of a main loop, in which every bytecode instruction from the VM is processed in three phases: *opcode decode*, *dispatch* and *execute*. The decode phase fetches the part of an instruction (opcode) that represents the instruction type. The dispatch phase uses this information to invoke appropriate handling routines. The execute phase, which is performed by the dispatched handling routine, performs additional fetches of operands and executes the semantics of the instruction. In the dispatch phase, Qemu splits one instruction of the VM into fewer simper instructions called *micro operations*. Each micro operation is implemented by a small piece of C code (handling routine). In an iteration of the main loop, Qemu fetches instructions of the VM up to the next jump or the one that modifies the static CPU state in a way that cannot be deduced at translation time. These instructions that are handled in a loop are composed of a Translated Block (TB). Each TB is indexed by the physical memory address of the first instruction of this TB. A 16 M byte code cache holds the most recently used TBs. It is completely flushed when it is full. At the beginning of a loop, Qemu uses the simulated PC to search the code cache and determines if the target TB has been translated. If yes, the host binary code of this TB is directly executed, then the dispatch phase is skipped. Otherwise, Qemu fetches the TB and goes to dispatch phase. Several optimization

techniques are employed in Qemu to improve its overall performance, e.g., condition code optimizations and direct block chaining. Besides the dynamic translator, the other two components of Qemu are emulated devices (mouse, VGA display and so on) and generic devices. They provide the emulated IO devices to VMs.

2.4 Related Work

Rootkit Detection

Integrity Verification [18] [32] [33] [34] is one popular rootkit detection approach that follows the spirit of Tripware [35] in protecting the file systems. It builds a baseline database for the measurable objects (e.g., text, static data) of the target guest OS by either storing the original data or computing the hashes. The detector periodically acquires the measurable objects from the current state of the guest OS and compares them with the baseline database to decide whether an intrusion occurs. Because it is hard to create baselines for dynamic data objects, this method can capture type I rootkits but not all type II rootkits.

Semantic Integrity Verification [26] extends the idea of integrity verification and creates a series of language-based specifications to describe the abstract models for low-level security-relevant data structure and the relationship between them. The detection is conducted through checking if kernel's state meets these specifications. This method can detect some type II rootkits, but it fails in the situation where the first logic predicate does not exist or is almost impossible to infer under an attack, e.g., contaminating the entropy pool of the Linux kernel [36]. Further, its specification generation depends on an expert's knowledge, so the overall effectiveness of this method is unknown.

Cross-view diff-based method [37] detects hidden objects (files, processes) by comparing a "low level" view with a "high level" view. One example is to use "ls /root" to show the file list under root directory from the user shell and run an agent to directly read the disk sectors containing "/root" to get another file list. If two file lists differs, then the different files are hidden. The agent, who obtains "low level" view in a detector, faces a dilemma: if it's not close to hardware, then the rootkits can intercept the reading operations and pollute its "low level" view; if it's close to hardware, e.g. accessing hard driver with "IN" and "OUT" instructions, the implementation complexity is increased as various HDD controllers need to be taken care of. On the other hand, rootkits can evade the detectors through temporarily disabling hidden behavior during their detection. Last, this method can only identify the rootkits aiming to hide system objects, but can not handle the rootkits with other attack vectors such as bypassing security software

Enforcing control flow integrity is s new rootkit detection approach proposed by [27]. The key idea of this approach is to intercept or inspect the kernel's dynamic branches (mainly function pointers) to ensure no illegal modification occurs. In [27], a type graph is constructed from parsing Linux kernel source, and the memory locations of function pointers are calculated from the global variables and type graphs. This approach is effective to thwart the rootkits that alter the kernel's control flow, but is not guaranteed to cover all the dynamic branches in the kernel, and also fails to deal with the rootkits that tamper with data other than function pointers.

Inference and Enforcement of Kernel Data Structure Invariants is used by system Gibraltar [31] to automatically detect rootkits that modify both control and non-control kernel data. The key technique is to externally observe the execution of the kernel during a training period and hypothesize invariants on kernel data structures.

Specifically, a graph of all kernel objects in the memory is created and the values of those objects' members are recorded at the training period, the dynamic invariant detector Daikon [76] is then applied to derive constraints on the objects' data. These constraints (invariants) are used as specification of data structure integrity during an enforcement phase: violation of these invariants indicates the presences of a rootkit. Because Gibraltar uses Myprinet PCI intelligent network card to fetch physical memory pages via the DMA communication, a rootkit who is aware of it can evade the memory acquisition by modifying system registers [77]. Moreover, Glbraltar doesn't address the ambiguous data types such as generic pointer, dynamic array and unions, it could miss significant portion of kernel objects [72].

Robust signatures for kernel data structures [71] is an automated mechanism that generate robust signatures for kernel data structures. Any attempt to evade the signature by modifying the structure contents will cause the OS to consider the object invalid. This mechanism doesn't need to create a baseline database for the rootkit detection. Instead, it is similar to cross-view diff-based method: running user-space program (like task manager) to get the user view of system objects (like processes) and then acquiring the kernel view of the system objects through searching the kernel memory and matching the signatures. Then, the comparison of two views can reveal the hidden objects. The signature of a kernel object is generated in three stages: 1. The target data structure is profiled to determine which fields are most commonly accessed by the operating system; 2. The most frequently accessed fields are fuzzed to determine which can be modified without causing a crash or otherwise preventing the structure from serving its intended purpose; 3. The dynamic invariant detector Daikon [76] is used to build a signature based on known-good instances of the data structure that depends only on the features that could not be safely modified during fuzzing.

This method is suitable to complex kernel data structures like EPROCESS. Unfortunately, many simple and small size kernel data structures may not have such robust signatures at all. In addition, the method also suffers the *coverage* problems in the profiling and fuzzing stages, which possibly leads to false positives.

Rootkit Prevention

Microsoft bolsters the driver signing technology and one of its purposes is to counter rootkit attacks. 64-bit Vista and Windows 2008 server require that only drivers signed by a trusted authority can be loaded to their kernels. Unfortunately, this rootkit prevention approach is weak. First, the current signing process is not designed to confirm the "intent" of signed code (i.e. good or bad), so malicious code is able to get a valid signature as well. And they, in turn, can disable Vista's code signing controls or bypass it through directly loading other code to the kernel [2]. Second, this approach is not plausible to most third-party driver developers as they have no easy way to debug their code on target systems, and development cycles become more complex, as each new version of the driver needs to be signed. Third, this approach can not deal with the thousands of legacy Windows drivers. Last, the bugs in the signed drivers can still be exploited by a hacker to take control of the kernel with rootkits [2] [3]. Because Microsoft hasn't applies this policy to 32-bit platforms, which own the majority of Windows users, the overall effectiveness of this approach is not proven so far, and time will tell the truth in the future.

NICKLE [28] and SecVisor [38] are two run-time intrusion prevention systems that leverage hardware virtualization extensions to prevent unauthorized kernel code from execution. Similar to the driver signing approach, both systems assume an authorization authority exists to classify a driver as trusted or distrusted. However

they do provide tighter security assurance than driver signing approach, because the latter cannot block zero-day kernel-code exploitations. Concretely, NICKLE employs a new scheme called memory shadowing, wherein the trusted VMM maintains a shadow physical memory for a running VM and performs real-time kernel code authentication so that only authenticated kernel code can be stored in the shadow memory. In addition, NICKLE transparently routes guest kernel instruction fetches to the shadow memory at runtime, which guarantees only the authenticated kernel code can be executed by VM. The downsides of NICKLE include the increased physical memory footprints of a VM and inability of supporting self-modified kernel code. SecVisor modifies the AMD's SVM to guard the page table of a guest OS, and enforces the W+X policy over the memory pages of guest OS, meaning that a page is either writable or executable, but never both. SecVisor achieves the same security guarantees as NICKLE at the cost of the requirement that the source of guest OS (Linux) need be modified.

HookSafe [73] is a hypervisor-based lightweight system that can protect thousands of kernel hooks in a guest OS from being hijacked. HookSafe achieves its functionality in two steps. First, an offline hook profiler component profiles the guest kernel execution and outputs a hook access profile for each protected hook. A hook access profile includes those kernel instructions that read from or write to a hook and the set of values assigned to it. Then, an online hook protector component takes hook access profiles as input and creates a shadow copy of all protected hooks and instruments hook access points instructions such that their accesses will be transparently redirected to the shadow copy. As such, these kernel hooks are relocated to a dedicated memory space and accesses to them are regulated with hardware-based page-level protection. This system suffers the limitation that the hook access profiles

are constructed based on dynamic analysis and thus may be incomplete. Further, HookSafe assumes the prior knowledge of the set of kernel hooks that should be protected, so some of kernel hooks are not included in their prototype. Last, HookSafe also impose non-negligible overhead (about 6%) to system performance.

Rootkit Analysis

Kruegel [39] and Limbo [40] use program analysis techniques to examine the innocence of a suspicious module. In particular, Kruegel performs program analysis with symbolic execution, a technique that simulates a program execution with symbols, e.g., replacing a variable's value with its name. Behavior specifications are created based on symbols to enforce the detection. This approach can be evaded by those rootkits that don't use the illegal symbols directly in the binaries: brute-force guessing a target memory address. In addition, it doesn't inspect the multiple execution path of a module and possibly misses some attacks. Limbo loads a suspicious module into an emulator and uses *flood emulation* to explore multiple running paths of a module. Behavior specifications of Limbo are automatically generated by applying data mining to a large number of malware samples. To increase the coverage of the examination, *flood emulation* doesn't keep the data states on paths, even including those deciding branches, so a module might behave abnormally in the emulator, resulting in inaccurate detection.

HookFinder [4] and HookMap [5] aim to identify the hooking behavior of rootkits. HookFinder performs dynamic taint analysis and allows users to observe if one of the impacts (tainted data) is used to redirect the system execution into the malicious code. On the other hand, HookMap is intended to identify all potential hooks on the kernel-side execution paths of testing programs such as *ls* and *netstat*. Dynamic slicing is

employed to identify all memory locations that can be altered to diverted kernel control flow. Unfortunately, hooking is only one aspect of rootkit behavior and both systems cannot provide comprehensive views of rootkit activities in a compromised system.

K-tracer [6] is a rootkit analysis system that automatically discovers the kernel data manipulation behaviors of rootkits including sensitive data access, modification and triggers. K-tracer performs data slicing and chopping on sensitive data in the rootkit trace and identifies the data manipulation behaviors. K-tracer cannot detect hooking behaviors of rootkits and is unable to deal with DKOM and hardware-based rootkits. An ideal rootkit analysis system should be able to handle a broad range of rootkits, including DKOM and hardware-based rootkits, and provide a complete picture of rootkit activities in a compromised system.

PoKeR [41] is another rootkit analysis system that shares the same design goal as this research: comprehensively revealing rootkit behavior. PoKeR is capable of producing rootkit traces in four aspects: hooking behavior, target kernel objects, user-level impact and injected code. Similar to Rkprofier, PoKeR infers the dynamic kernel object starting from the static kernel objects. However, PoKeR only tracks the pointer-based object propagation, while ignoring the function-based object propagation. This limits PlKeR's capability of identifying kernel objects. Furthermore, the function call and hardware access monitoring features are not offered by PoKeR.

KOP [72] is a kernel analysis system that can map dynamic kernel data with nearly complete coverage and nearly perfect accuracy. It maps all the dynamic kernel objects by performing a complete traversal of the memory, starting from a set of globally well-defined objects and following each pointer reference to the next object, until all

have been covered. KOP addresses three ambiguous data types: generic pointer, union and dynamic array. Concretely, KOP applies inter-procedural points-to analysis to compute all possible types for generic pointers; a pattern matching algorithms is chosen to resolve the type ambiguities of unions; it recognizes dynamic arrays by leveraging knowledge of kernel memory pool boundaries. The output of KOP is an object graph that contains all the identified kernel objects and their pointers to other objects. Although KOP doesn't directly profile or detect kernel malware, it provides the solid foundation (kernel data maps) for other systems to fulfill these functionalities. Hackers can disrupt KOP's traversal by polluting the kernel memory. For instance, they may intentionally break the internal structure of key kernel objects by tampering with the values stored at pointer fields. As a result, the KOP's traversal might incorrectly identify these objects due to pointer field mismatches. Another shortage of KOP is that it may not be able to detect non-global transient kernel objects that only live within a function.

Binary Translation

Several recent projects have borrowed techniques from dynamic optimization to rewrite programs on the fly; such techniques allow for fine control of program execution. Valgrind [42] is a powerful framework for dynamic rewriting of Linux/x86 programs, which can be adapted to security application development. Valgrind's rewriting uses a simplified intermediate language, sacrificing performance for ease of development. Strata [43] is a more security-oriented tool that achieves lower overheads (about 30%) while enforcing targeted security policies such as system call interception. Kiriansky's program shepherding [44] is build based on the dynamoRIO dynamic translation system. Their work concentrates on preventing attacks on a

programs' control flow, as an efficient and transparent means to prevent stack and function-pointer-smashing vulnerabilities from being exploited. All these binary translation systems impose a high performance penalty and can't be directly used as an inline prevent system. In addition, they need to run in the same OS space as malware, and cannot resist malware's attacks.

Demand Emulation

Ho [45] first proposed the concept of Demand Emulation that can be used to solve the security problems. His system modifies the emulator's hardware support to enable data tainting at the system level. The system is built on Qemu and Xen VMM, and its main application is to detect and prevent malicious code injection by tracking data received from the network as it propagates through the VM. In the system, the incoming network data are tainted and traced during their lifespan, which requires relatively heavy emulation. Moreover, as Xen and Qemu each have their own fault and interrupt handlers, the Ho's system requires a fault or interrupt issued in emulation mode must first be trapped and processed by Xen's handler and then passed to emulator's handler.

CHAPTER 3

KERNEL INTRUSION PREVENTION

In this chapter, we present a kernel intrusion prevention system that tracks LKM-based rootkits at a granite level by using *on-demand emulation*, a technique that dynamically switches a running system between virtualized and emulated execution. The basic idea is to sandbox a suspicious loadable kernel module in an emulator and to assure its goodness by enforcing a group of well-selected security policies. Based on open source software Qemu and Kqemu [82], we designed and implemented a software system, named DARK that uses on-demand emulation to provide a powerful defense against kernel malware. In DARK, when a rootkit tampers with a kernel object or hardware object, its illegal behavior is captured and blocked. In the meanwhile, VM emulation takes place only at time that a suspicious module is executed, and the most operations of the VM are performed in the virtualization mode. Thus, the substantial execution overhead caused by emulation is avoided. Our contribution in this work includes:

1. Identification of non-integrity-violation rootkits that can escape kernel integrity verifiers.
2. Implementation of a novel rootkit prevention system based on on-demand emulation to sandbox a suspicious kernel module.
3. Creation of a group of security policies to detect and block all rootkits we collected.

3.1 Motivation

Kernel run-time protection mechanisms can be categorized as prevention and detection. Previous run-time rootkit prevention approaches [28] [38] focus on protecting the benign kernel code and thwarting malicious kernel code. One key issue here is how to determine the goodness and trustworthiness of a piece of kernel code. Unfortunately, previous approaches failed to give in-depth analysis of this problem and just simply assume it is *a priori* knowledge to end users or protection systems, which is not true in practice. To the date, there is no such commodity operating system that strictly controls the kernel code loading based on both goodness and trustworthiness of kernel code. Even Microsoft's driver code signing [1] is just employed for the identification of driver authors, but not for assuring the goodness of signed drivers [2]. Even though, the effectiveness and robustness of this mechanism are still a question mark [2] [3]. In the end, people have to make decision on whether to install a useful but potentially unsecure driver, which is a challenge that is not addressed by previous approaches.

Run-time Rootkit Detection Methods proposed by researchers can be divided into two categories: specific rootkit detection and generic rootkit detection. Methods in the first category focus on capturing a specific type of rootkit. For example, *Cross-view diff-based* method [37] just targets those rootkits that conceal disk objects (files and registries); Lycosid [29] is intended to discover hidden processes only. On the contrary, methods in the second category are designed to counter broad types of rootkits. To best of our knowledge, the most generic rootkit detectors known to the public are kernel integrity verifiers [18] [26] [27] [32] [33] [34]. Kernel integrity

verifiers concentrate on examining the states of some kernel objects to ensure that illegal tampering of these objects don't occur. They are effective to defeat integrity-violation rootkits. Unfortunately, theses kernel integrity verifiers suffer two fundamental weaknesses: incompleteness of assuring the integrity of dynamic kernel objects; incompetence at detecting non-integrity-violation rootkits, like confidentiality-violation rootkits and hardware-exploiting rootkits. These two weaknesses are discussed in detail as below.

Dynamic Kernel Objects

Most kernel rootkits are implemented in the form of kernel modules (drivers). Hence, they share the same virtual memory environments as operating system. No matter whether a kernel object (structure, list, text and so on) is exported or not by the OS, a rootkit can always directly access and tamper with it after being loaded to the kernel. In fact, direct kernel object manipulation (DKOM) is one prevalent technique employed by rootkit writers [47]. A kernel object could reside on either permanent memory area (text, dss) or transient memory area (stack and heap); its content could be constant or changeable. A kernel object is static if its memory address is permanent. Otherwise, this object is dynamic. Defending a static kernel object is straightforward, as its location and content is relatively easy to identify. On the other hand, protecting a dynamic object could become challenging due to the following four reasons. First, in comparison with static objects, the population of dynamic objects is much larger, and enumerating all dynamic kernel objects at any time could be impractical. Second, since integrity verifiers have to wake up to work periodically, they miss catching lots of short-lived dynamic objects, e.g., local variables in stacks. Third, a detector's recognition of dynamic objects can be attacked by rootkits so that those objects are invisible to the detector. For examples, rootkits can alter the page

table to hide kernel objects from detectors, or remove an element from a link list to make it untraceable. Last, the content of a kernel object can be unpredictable and detectors are unable to differentiated good and bad values. One such example is the entropy pool of the Linux, which can be manipulated by rootkits to compromise Linux Pseudo-Random Number Generator (PRNG) [36]. In summary, kernel integrity verifiers cannot assure the integrities of all dynamic objects in a kernel.

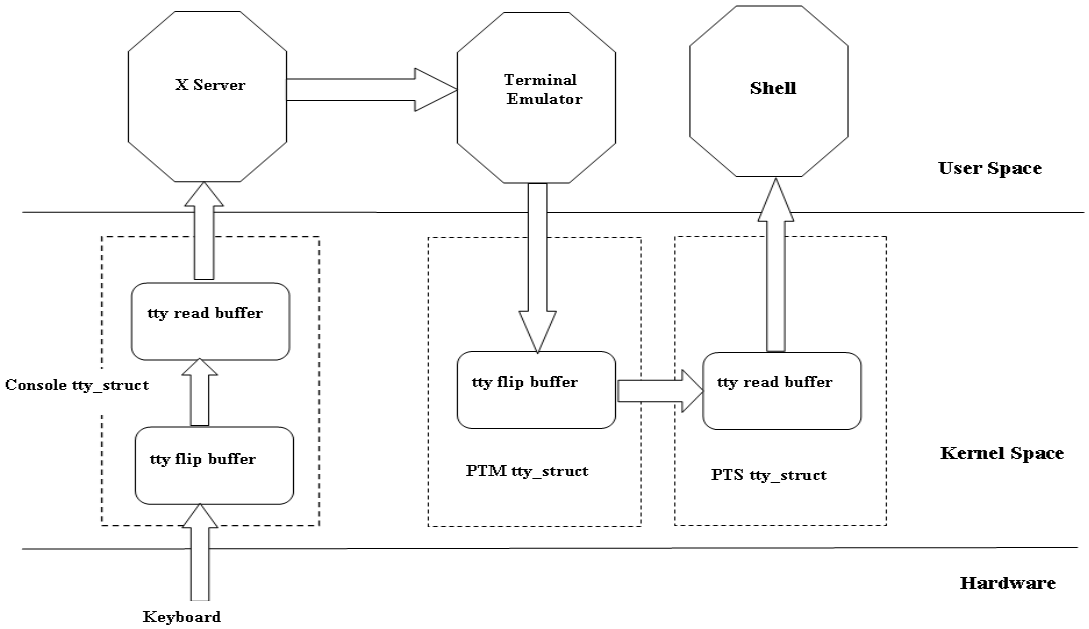


Fig. 1. Keystroke data flow in Linux desktop

Non-integrity-violation Rootkits

Non-integrity-violation rootkits refers to those that launch attacks while not manipulating any kernel objects, so kernel integrity verifiers can't catch them. One type of non-integrity-violation rootkits are hardware-exploiting rootkits [10] [48] [49] [50] [51], which misuse a hardware feature or configuration to achieve their goals. Another type of non-integrity-violation rootkits: confidentiality-violation rootkits. They only break the kernel data confidentiality while preserving the data integrity. One class of candidates for the confidentiality-violation rootkits is data theft rootkits,

e.g., keylogger and network sniffer. Next, we demonstrate one confidentiality-violation rootkit: a Linux keylogger (called darklogger) that can sniff the keystroke without illegally changing any kernel object.

Today, common Linux desktop environments like Gnome and KDE use the X window system to manage terminal services: interacting with keyboard and mouse, drawing and moving windows on the screen. The key data flow in a typical X window system is shown in figure 1. On the X server, the key reading path from keyboard to user space consists of at least two threads working in tandem: a top thread originating from a user process that issues read requests, and a bottom thread originating from the interrupt service routine that reads the key data from the keyboard. Two kernel buffers, *tty flip buffer* (tty_struct.flip.char_buf) and *tty read buffer* (tty_struct.read_buf), store the key data (interpreted by keyboard driver) and provide the synchronization between the top thread and the bottom thread. When the top thread asks for data and the *tty flip buffer* is empty, the thread goes to sleep; when the bottom thread fills new key data to the *tty flip buffer*, it awakes the top thread who copies the new data from the *tty flip buffer to tty read buffer* and then to user space. In figure 1, when a key is generated by keyboard and travels to the shell, it may be kept in four kernel buffers. By adding hooks or patching code, traditional keyloggers hijack the control flow of kernel's processing key data. Darklogger takes a passive approach based on the observation that *tty read buffer* is a large-size circular buffer and a char data, representing a key, in the buffer is not wiped off until the head pointer of the buffer moves back to its location, where a new char data is written. Since human's keystroke speed is relatively slow (less than 30 characters/second) and the size of *tty read buffer* is large (4k), it takes more than 2 minutes to fill up the entire buffer. Darklogger is a kernel thread that wakes up every 10 seconds to read the

tty read buffer and acquire all key data. Based on the positions of the head and tail pointers in the buffer, Darklogger is able to extract the key data of the last period. Because Darklogger just uses the legal kernel APIs and doesn't maliciously hook any function or modify any kernel data object, it can evade all kernel integrity verifiers.

Following the spirit of sandboxing program [44], DARK captures the interactions between a rootkit and the rest of a kernel. The kernel objects visited (memory read, write and function call) by a rootkit are recorded and analyzed regardless of their locations, lifespan and contents. To DARK, the rootkit defense is an access control problem and its success depends on the effectiveness of the security policies.

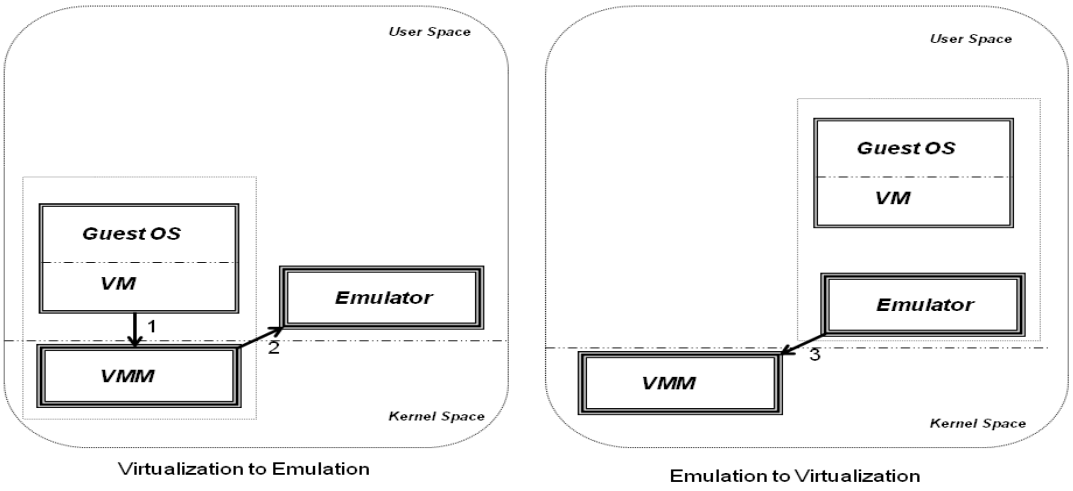


Fig. 2. An on-demand emulation system

3.2 On-demand Emulation

Virtual Machine Monitor (VMM) and emulator are two types of hypervisors that support and manage multiple virtual machines (VM). A VMM seeks to achieve high performance by directly executing most instructions of a VM on the host (physical) CPU. In contrast, an emulator translates each VM's instruction to the host instructions

so as to provide different types of virtual CPUs to its VMs, paying the cost of poor performance. Due to their deep inspection capabilities, some researchers use emulators to perform various security related tasks, e.g. malware identification and analysis [40].

DARK is a hybrid system that combines the strengths of VMM and emulator to offer better system security and performance. It contains three components: a VMM, an emulator and a VM where a guest OS is installed. In virtualization mode, the virtual machine runs on top of the VMM to gain nearly native speed. When a suspicious module is to be executed in the VM, the VMM is informed to take control of the VM. Then, the VMM collects the virtual CPU state and status data of memory management unit (MMU), and sends them to the emulator. Thus, DARK is switched to emulation mode. Once receiving the VMM's virtual CPU state, the emulator restores the VM's operation and start monitoring the module's activities and enforcing the security policies accordingly. When the execution of module code is completed, the emulator suspends the VM and passes its control back to VMM with the current virtual CPU state and MMU status data. The VMM restores the VM and DARK is switched virtualization mode. The emulation is required only when the target module is executed, and most of VM codes still run on VMM.

3.2.1 Design

The primary task of the on-demand emulation is to trap the module execution in a VM. However, a module may have many non-privilege instructions and their executions in a VM cannot trigger exception or interrupt, which is only way of transferring the control from VM to VMM in a virtual machine system. DARK addresses this problem by exploring the paging mechanism of operating systems. A

present bit in the page table entry indicates whether a virtual page has been assigned a physical page frame. When a CPU accesses a virtual page whose present bit is 0, the memory management unit (MMU) generates a page fault. Then, an interrupt routine is invoked to allocate a physical page frame and copy the page data from the swap area or disk file (demand paging) to this physical page frame. As Linux never swaps kernel codes to disk, the present bits of kernel code pages are always set to 1. DARK can trap a module by clearing the present bits of its code pages in the virtualization mode. Later, when the module is to be executed, VM issues a page fault. Thus, the VMM of DARK intercepts the exception and passes the control to emulator, who sets those *present bits* back to 1 and starts executing and monitoring the module in the emulation mode. To maintain the integrity of the existing page fault mechanisms, the page fault handler of guest OS should be modified to properly deal with these manipulated page faults.

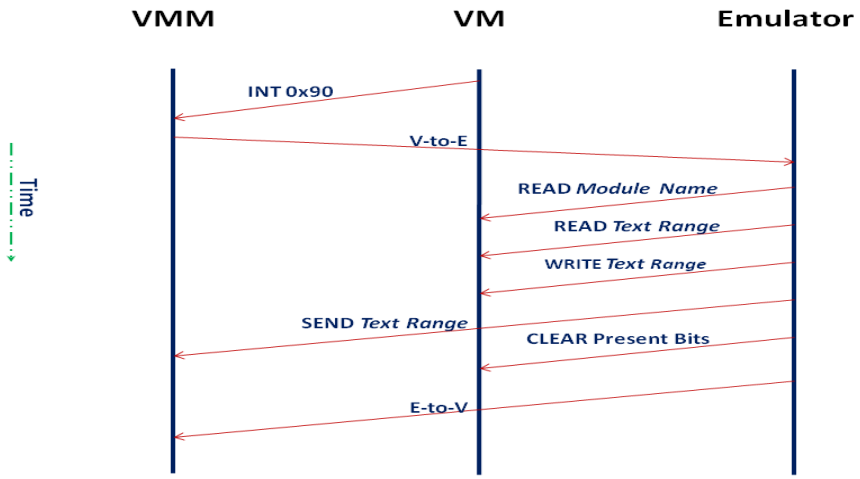


Fig. 3. Partial on-demand emulation process

Before loading a module to the guest OS, the DARK user decides whether to monitor the module or not. If yes, the emulator is notified of the module name. To change the present bits of the module before its execution, the guest OS issues a software interrupt through instruction "*int 0x90"*. The VMM catches the interrupt, and hands it over to the emulator. Then, the emulator fetches the module name from the VM image and compares it with the one defined by DARK user to decide if the current module is right target. If two names are different, DARK gives up monitoring and switches back to virtualization. Otherwise, DARK kicks off the monitoring with the following steps. First, the emulator queries the text (code) range of the target module from the module list of the guest OS, and sends it to the VMM. Then, it clears the module present bits and transfers the control to VMM, forcing the system into the virtualization mode. Later, when the module is to be executed, the VM generates a page fault, which is trapped to the VMM. The VMM uses the text range of the module to identify that the faulty instruction comes from the target module, and transfers the VM control to emulator. After setting the module present bits to 1, emulator restores the VM sessions and starts the monitoring process again. In this way, DARK moves the VM control between VMM and emulator back and forth depending on if the VM executes module code. Figure 3 depicts this on-demand emulation process. When the module is unloaded, DARK turns off on-demand emulation by cleaning up their monitoring records and set the corresponding present bits in the VM to 1.

3.2.2 Implementation

DARK is built on Qemu and Kqemu, who run on any X86 CPU regardless of the hardware virtualization support. Both guest OS and host OS are Redhat Linux. As described in Chapter 2, Qemu is a hardware emulator that uses binary translation to simulate processor and peripherals, while maintaining a reasonable speed. Kqemu is a

kernel module that works with Qemu to provide virtual machine monitor functions. In the full virtualization mode of a Qemu/Kqemu system, all user-mode instructions and some kernel-mode instructions of a VM can be directly executed on the host CPU. For security reasons, the kernel-mode instructions for memory accesses in the VM have to be intercepted and interpreted by Kqemu. This is done by clearing the global descriptor table (GDT) and local descriptor table (LDT) when the VM runs in kernel mode. Thus, any kernel-mode memory access in the VM will cause a general protection fault. Kqemu captures these faults and interprets the instructions in the kernel. Because Kqemu needs Qemu to handle some corner cases such as interpreting a HLT instruction, some components of the on-demand emulation framework are already available in orginal Qemu and Kqemu software. To enable module tracking, DARK modifies the switch control code of the existing on-demand emulation framework. In particular, DARK adds the following business logics to the interrupt handler and V-to-E (virtualization to emulation) control code (in common/module.c and common/kernel.c) of Kqemu:

1. *If an interrupt vector number is 0x90 or 0x91, does emulation switch.*
2. *For a page fault, if the faulty instruction address is within the text range of target module, does emulation switch.*

Moreover, we add one boolean variable to Qemu's E-to-V (emulation to virtualization) control code to ensure that virtualization switch is disabled when the current instruction is from the target module and vice versa.

In addition, we instrument the guest OS kernel (Linux version 2.4.18): adding two assembly instructions to *sys_init_module* and *sys_delete_module* functions in kernel/module.c. The first instruction issues a software interrupt 0x90 before loading a module; the second one issues the interrupt 0x91after unloading a module. DARK obtains a module name by reading the module descriptor from the kernel module list.

Further, we modify the Linux module loader (insmod.c) to put the module text range in the *runsize* and *kernel_data* fields of the module descriptor, which allows DARK to read the text range later. In Linux, all processes share one kernel page table that can be accessed from the kernel master page global directory *swapper_pg_dir*. We use this global variable to locate the page table entries of the target module and rewrite the module *present bits* as described in Section 3.1. Last, we alter the page fault handler of the guest OS such that it ignores the page faults caused by the target module execution.

3.3 Security Policy

DARK does not aim to build perfect security policies to catch all rootkits. In fact, modern operating systems are not designed to be traceable and verifiable, so the creation of such "perfect" policies may be impossible. Rather, similar to SELinux [30], DARK provides a policy framework that gives security administrators the flexibility to write their own security policies. To demonstrate the effectiveness of DARK, we compose a group of security policies that are good enough to detect most existing Linux rootkits and raise the bar for future kernel exploits.

3.3.1 Policy Framework

DARK treats the rootkit detection as an access control problem: a malicious module needs to illegally access another part of kernel to perform the attack. DARK's security policy is composed of a group of access control rules whose format is given in table 1.

Table 1. DARK's rule format

Subject	Operation	Object	Action
{module a, b, c ...}	*{read, write, call}*	*{hardware objects, kernel objects}*	*{reject, alarm}*

In table1, *subject* is a module that is to be monitored. A module's *home space* contains: object (code and global data) section, stack and heap. Any instruction issued from a module space is regarded as a representative of this module, and should be monitored. Note DARK can apply various policies to different modules, which is discussed later. *Operation* indicates the way that a module interacts with the rest of kernel. DARK tracks three types of operations performed by a module: *read, write* and *call*. First two are memory access operations*; call* is an act that a module invokes functions exported by OS and other modules. Although a module may influence the kernel objects in other means, e.g. return of an external call, creating a system exception, these three operations are sufficient for DARK to detect the rootkits we know.

Object refers to those system resources and services accessed by a module. Two types of system objects are included in DARK: hardware objects and kernel objects. The former contains dedicated registers, IO ports and IO mapped memory. Many of these hardware objects are crucial to system security. For example, the register IDTR holds the linear address of interrupt descriptor table which is used by CPU to transfer an interrupt to the corresponding Interrupt handler. Hijacking this register allows hacker to amount various attacks, e.g. installing a virtual-machine-monitor-based rootkits [10] [51]. Kernel object is a software concept, and one kernel object is a group of kernel data or code that is semantically meaningful to software developer, like a structure, an integer variable and a function. Several kernel objects, e.g., system call table, are common attack targets to Linux rootkits.

In DARK, a policy rule that contains a hardware object is called *system rule*; a rule whose object field is a kernel object is called *kernel rule*. A hardware object has only

one representation in DARK, and it may be a register name, or IO port number or memory address. One kernel object has two representations: one is a software-level representation such as variable names and function names, which is used by DARK users to make policies; the other is a hardware-level representation and it is the memory address of the corresponding software object. Since DARK enforces policy at the hardware level, for a kernel rule, it's necessary to translate its software-level representation to the hardware-level representation, which is called *policy translation*.

DARK's kernel rules may contain both *static kernel objects* and *dynamic kernel objects*. A static kernel object's memory address is determined when the kernel is build, so this object's location is fixed all the time, e.g., system call table. Conversely, a dynamic kernel object's location can only be decided at run time, e.g., a process' page table. A kernel rule containing a static object is called *static kernel rule*; a kernel rule containing a dynamic object is called *dynamic kernel rule*. Unlike static kernel rules whose policy translation can be performed before a VM is powered on, policy translation of the dynamic kernel rules has to be postponed to run time.

DARK takes two actions on a policy violation: *reject* and *alarm*. *Reject* denotes that DARK immediately stops executing the target module and prevents any further damages. In Linux, removing a module is more complex and risky than deleting a process from the system, and the former can corrupt the OS's operation integrity and reliability. Current implementation of *reject* action terminates the VM, and writes a warning message to a log file on the host OS. Granular failure remediation is of the future work. DARK's *alarm* action only requires generating the logging messages instead of turning off the whole system. Determination of a *reject* or *alarm* action for a rule is based on the consideration of multiple factors: severities to system security, reliability and stability. For those attacks that not only compromise the security but

also greatly degrade the system reliability and satiability, *reject* should be the choice, e.g., runtime patching of the kernel text; For other attacks, terminating the current system operation is not necessary, and alarming is probably sufficient to enable defense measures such as sniffing network traffic.

Table 2. DARK's kernel rules

ID	NAME	OPERATION	GLOBAL VARIABLE OR FUNCTION	DATA TYPE	ACTION	DYNAMIC	OPTIONAL
1	Console TTY Buffer	Read	console_table	tty_struct	Alarm	No	No
2	Exception Table	Write	__start___ex_table	Exception_table_entry	Alarm	No	No
3	GDT table	Write	gdt_table	Array	Reject	No	No
4	IDT table	Write	idt_table	Array	Reject	No	No
5	Kernel Text	Write	_text	-	Reject	No	No
6	MM List	Write	init_task	mm_struct	Alarm	Yes	No
7	Module List	Write	module_list	Module	Alarm	Yes	No
8	Module Text	Write	module_list	-	Reject	Yes	No
9	Netfilter Hooks	Call	nf_register_hook	-	Alarm	No	Yes
10	Page Table	Write	init_task	-	Reject	Yes	No
11	Proc Dir Entry List	Write	proc_root	proc_dir_entry	Alarm	Yes	No
12	Proc Inode Ops List	Write	proc_root	Proc_inode_operation	Alarm	Yes	No
13	Proc File Ops List	Write	proc_root	Proc_file_operation	Alarm	Yes	No
14	PTM TTY Buffer	Read	ptm_table	tty_struct	Alarm	Yes	No
15	PTS TTY Buffer	Read	pts_table	tty_struct	Alarm	Yes	No
16	Socket Buffer List	Read	skbuff_head_cache	sk_buff	Alarm	Yes	Yes
17	Syscall Table	Write	sys_call_table	Array	Reject	No	No
18	Task List	Write	init_task	task_struct	Alarm	Yes	No
19	Task State Segment	Write	init_tss	Array	Reject	No	No

3.3.2 Established Rule

DARK's policies are constructed based on common knowledge of the OS security and observation of attack patterns of the existing rootkits. Total 19 kernel rules are created and shown in table 2. Among them, four *read* rules and one *call* rule are used to address the data theft rootkits as discussed in 2.2. The rest fourteen *write* rules deal with kernel integrity. Eleven dynamic rules employ seven global variables as the starting points of policy translation. Among them, six global variables are single/double linked lists and the other one (*proc_root*) is associated with binary tree data structure. Note these global variables should be write-protection as well. Otherwise, rootkits may modify the variables to hinder the policy translation. We find that early-stage rootkits tend to manipulate the static kernel objects such as system call table and kernel text. These objects are critical to the system reliability and stability, any illegal modification of them should be rejected at once. Kernel objects contained in Rule 5 and 17 are such examples. On the other hand, some kernel rules are devised to counter the threats in the future, while not being hit by any existing Linux rootkit. For example, it's reported that some Windows rootkits tamper with the kernel memory management system to hide some kernel objects. It can be foreseen that hackers may apply the same technique to Linux rootkits down to the road. Rule 6 and 10 are designed to achieve such purpose. Rule 9 and 16 in table 3 are optional, because many normal networking drivers may violate them and enforcing these rules possibly generates false alarms. The usages of optional rules depend on user's knowledge to the target modules. Beside kernel rules, we create 11 system rules, and most of them are applied to special system instructions that handle critical system-level functions, e.g., SGDT and WRMSR.

Table 3. DARK's system rules

ID	Name	Operation	Hardware Object	Action	Instructions
1	BIOS	Write	BIOS ROM	Reject	MOV
2	System Cache	Write	L1, L2 Cache	Alarm	INVD, WBINVD
3	Control Register	Write	CR0, CR3, CR4	Reject	MOV CRn
4	Debug Register	Write	DR0-DR7	Alarm	MOV DBn
5	IO Port	Read/Write	IO Ports	Alarm	IN, OUT
6	IDT Register	Write	idt register	Reject	LIDT
7	GDT Register	Write	GDT Register	Reject	LGDT
8	MSR	Write	MSR	Alarm	WRMSR
9	System RAM	Write	System RAM	Alarm	MOVE
10	TLB	Write	TLB	Alarm	INVLPG
11	TR Register	Write	TR Register	Reject	LTR

3.4 Enforcement

DARK stores the security rules to a local file called policy.dat. This file contains the system rules, static kernel rules and software-level dynamic rules. When a VM is started, DARK forks a thread that performs three tasks: 1. loading the policy.dat to the RAM; 2. periodically translating dynamic kernel rules to the hardware-level representation; 3. transforming all memory-access rules to the hash-table based rules as discussed in Section 5.1. This thread stores all the rules to several global variables, which are used to enforce the policy at run time.

When a suspicious module is to be executed, the emulator takes control of the VM and begins policy enforcement. Concretely, DARK intercepts all memory access instructions and some system instructions at the binary translation of Qemu. Note an alternative method is to change the Qemu's simulated MMU to capture the memory accesses. However, this method cannot enjoy the benefit of code caching and suffers a larger performance penalty. For each of the monitored instructions, DARK checks the corresponding rules. If an instruction hits a rule, DARK takes the action defined in the rule. For *alarm*, DARK writes one warning message to the system log on the host

machine. The message includes the module name, the instruction's address and the rule id. For *reject*, DARK generates an alarm and then powers off the VM by terminating the current Qemu process.

3.4.1 Hash Table

The data structures that hold memory access rules should be selected prudentially, as an inappropriate data structure might hurt system performance. DARK's memory access rules are initially defined as a series of memory intervals. One memory interval, like (0xC03254fa, 0xC03256a0), is called one *memory bucket*. Some dynamic rules, like socket buffer descriptors, comprise a large amount of memory buckets. If they are stored in link lists, DARK needs to traverse thousands of memory buckets (with various sizes) to inspect one instruction in a linear time of n. We present a data transformation method that converts a link list of memory buckets to two hash tables. Since hash table lookups takes O(1), it can significantly reduce the enforcement overhead.

```
VOID convert_bucket (ULONG start_address, ULONG end_address, int bit) {
        ULONG   mask, page_frame, key, current_address;
        struct value_struct *val;

        mask = (1 << bit) – 1;
        page_frame = 1 << bit;
        current_address = start_address;
        while (1) {
                key = current_address & (~mask);
                val = (value_struct*)malloc(sizeof(value_struct));
                val.start_offset = current_address & mask;
                if (current_addess + page_frame < end_address) {
                        val.end_offset = 0;
                        insert_hash_entry(bit, key, val);
                        current_address += page_frame;
                }
                else { //current_address moves to the last page frame
                        val.end_offset = page_frame – (end_address & mask);
                        insert_hash_entry(bit, key, val);
                        break;
                }
        }
}
```

Fig. 4. Source code of the memory bucket transformation routine

Similar to the OS concept of 32-bit page frames, DARK uses 10-bit and 5-bit page frames in the transformation. The memory interval of a bucket is broken into multiple 10-bit or 5-bit page frames and each page frame has one entry in a hash table. Two hash tables store 10-bit and 5-bit page frame rules respectively. Figure 3 lists the C implementation of the converting routine. The selection of 10 and 5 bit page frames is based on the observation that most memory buckets created by DARK are either large (at the page level) or small (less than 200 bytes). This division ensures that each hash table is not overwhelmed due to hash conflicts. Given a target memory address, DARK first computes its 5-bit page frame address by removing last 5 bit of the memory address, and searches for the frame address on the 5-bit hash table; if not found, it then does the same check for the 10-bit hash. Thus, only two bit operations and two hash table lookups are needed, at most.

3.4.2 Code Cache

To reduce the emulation overhead, DARK takes advantage of the performance optimization in Qemu. The key technique is to cache the translated code sequences so that they can be directly executed in the future. Each sequence of instructions ending with a single control transfer instruction is called a block. Qemu translates a block in each main control loop and places the translated block to a code cache. All the translated blocks are organized as a hash table and a cached block can be found fast. A block can be linked to another one if it doesn't contain the indirect branches, avoiding the extra loop cost. DARK only performs the security check at binary translation, so once a block of code is put into the cache, DARK doesn't examine it any more. Finally, when the code cache is full, Qemu simply purges all blocks in the cache and refills the cache with new blocks. Since DARK’s emulator only caches small-size module code, the chance of overflowing the cache is small.

3.4.2 Security Log

DARK provides the logging capability that keeps record of the interactions between a module and the rest of the kernel. The log includes: memory write and read, function call and IO operations. For memory read and write, DARK prints out the instruction address, and target memory address and content. For function invocation, DARK records the function address, calling instruction address, the first two parameters and return value of the function. However, parameter semantics of a function are unknown, so DARK logs the first 16 bytes in the stack parameter area of the function. Note that DARK only logs the external memory accesses and function invocation. In addition, we create a tool that interprets log records, identifies all heaps that are assigned to the module, and removes them from the log. Combining this logging capability with Qemu's snapshot can provide an abundant data source for forensic analysis.

3.5 Evaluation

This section presents the empirical results of the DARK system. The evaluation is composed of two subsections. In the first subsection, the functional effectiveness of DARK is investigated: whether the security policies are made properly in terms of false positive and false negative detection rates. Then, we conduct the performance evaluation and study the performance impact of on-demand emulation on the VM. DARK is built based on the QEUM 0.8.2 and KQEMU 1.3.0prell. All the experiments are performed on a Dell machine with Intel P4 CPU (2.8 GHz) and 1 GB RAM. The host OS is Fedora Core 5.0 and the guest VM was assigned 256M RAM and 6G hard drive; Guest OS is Red Hat Linux 8.0 with 2.4.18-14 kernel.

Table 4. DARK's rootkit detection result

ROOTKIT	FUNCTION					TYPE	HIT KERNEL RULE		ACTION
	HID	PE	REE	REC	NEU		Load	Operation	
Adore	X	X				I	17	18	Reject
Adore-ng	X	X			X	II	7, 12, 13	18	Alarm
Adore-ng (hidden)	X	X			X	II	7*, 12, 13	18	Alarm
Darklogger				X		II		15	Alarm
Exception		X			X	I	2	18	Reject
Fileh-lkm	X					I	17		Reject
Hookstub		X				I	4	18	Reject
Hp	X	X				II	18		Alarm
KIS	X		X			I	17		Reject
Knark	X	X	X			I	17	18	Reject
Linspy2				X		I	16		Reject
Nfsniffer				X		II	9	16	Alarm
Nushu					X	II		16	Alarm
Pizzaicmp			X			II	9	16	Alarm
Prrf	X	X				II	11, 12, 13	18	Alarm
Sebek				X		I	7, 17		Reject
Srootkit	X					I	5		Reject
Vlogger				X		I	17	14	Reject
Vlogger (local)				X		II		1	Alarm

3.5.1 Security

In this experiment, we collect 18 rootkits that cover a wide range of attacks. Among them, there are 10 type I rootkits, 8 type II rootkits, 8 HID rootkits, 7 PE rootkits, 3 REE rootkits, 5 REC rootkits and 3 NEU rootkits. In addition, one rootkit from [52] is devised to attack the hardware resources (system BIOS). Unfortunately, the Qemu's BIOS is not updatable, so the rootkit cannot be successfully installed to the test VM. The other 17 rootkits are listed in table 4. A rootkit may have several operation modes and different modes may use different attack tactics. For example, with the technique described in [16], Adore-ng can optionally hide itself into a benign module, forming a "combo" module. We test the regular Adore-ng and hidden Adore-ng separately.

To comprehensively understand rootkit behaviors, we run several Linux utilities like *ls, ps, netstat* and *ssh* to verify whether a rootkit works as expected after its installation. Moreover, when a rootkit violates a *reject* rule, we intentionally instruct

DARK not to shutdown the guest VM and make the rootkit continue to run until all testing utilities are finished. Thus, we can catch all security rules that the rootkit hits.

The test result in table 4 suggests that DARK is able to detect all the rootkits with the security rules in table 4. Some rootkits violate multiple rules at loading stage and operation stage. System call table (rule 17) and task list (rule 18) are primary kernel objects that rootkits target on. Several type I rootkits hijack system call table to hide user-space objects or steal private data. IDT table and kernel exception table are another two static kernel objects that the rootkits tamper with in the test. To type II HID rootkits, proc file system provides exploitable kernel objects that are alternatives to system call table: two such rootkits (adore-ng and prrf) alter the relevant data objects of the proc system to hide processes and network connections. All the PE rootkits modify the user id and group id in the *task_struct* objects to raise a process' privilege level. Another observation is all rootkits are captured at the loading stage except the Darklogger and Nushu. As we pointed out before, Darklogger is a non-integrity-violation rootkit and does not illegally change any kernel object in the kernel. It just creates a kernel thread and initializes some data structures at the loading stage. Yet, its reading the PTS buffer is caught by DARK at the operation stage. Nushu manipulates the packets from/to local network adapters by indirectly registering hooks to the kernel through the function *dev_add_pack*. Because this function is not defined in table 3, Nushu escapes the loading-stage inspection. But DARK detects the intrusion when it reads socket buffers at the operation stage. Note that powerful kernel integrity verifiers are still likely to catch the Nushu due to its hooking behavior.

In the experiment, Adore-ng is embedded in the module *iptables_filter* to create a combo module. By comparing the hidden Adore-ng with the regular Adore-ng, we

find that they hit the same set of rules. However, the combo module can not be unloaded from the kernel even after we flush the iptable rules and stop the iptable service. After further investigation, we found the reason. Both hidden Adore-ng and the regular Adore-ng modify the kernel module list, which is a list of *module* objects. The regular Adore-ng changes the *next* fields of the previous and next *module* objects with the purpose of hiding itself, while the combo module alters the *uc.usecount* field of the current *module* object to persist its existence in the kernel. Vlogger is also tested in two operation modes. Although the regular mode offers more powerful features than local mode, the latter turns out to be stealthier: it only alters the dynamic kernel objects and is a type II rootkit.

Table 5. DARK's false positive test

Type	Driver	Action
net	8390	Pass
	Ne2k-pci	Pass
	Ip_tables	Pass
	Iptable_filter	Pass
fs	Autofs	Pass
	jbd	Alarm
cdrom	Cdrom	Pass
ide	Ide-cd	Pass
input	Input	Pass
	mousedev	Pass
Sound	soundcore	Pass
crypto	cryptoapi	Pass

To estimate the false positive rate of the detection system, we choose 7 categories and total 12 drivers from the Linux source, and execute them in the DARK system. When we test the network drivers, we inactivate the optional rules 9 and 16 to avoid the false alarms. The test result indicates that 11 of 12 drivers pass the test. The failed module is jdb and it is a journaling block device driver used by Ext3 file system for data recovery. This driver alters the *journal_info* filed of two process' *task_struct* objects, leading to the violation of rule 18. This false alarm implies that the rule 18 is

too restrictive and should be revised to only include the sensitive fields that task list members. But, on the other side, this violation doesn't incur the system termination and we believe that overall quality of the security rules is good.

3.5.2 Performance

Performance evaluation is intended to measure the impact of on-demand emulation on overall system performance. The module *iptable_filter* from Linux source is chosen to be monitored. First, this module operates at the kernel network stack, which is one of major attacking targets to rootkits. Second, running this module in emulation mode is expected to only degrade the performance of the network subsystem in the kernel, and other subsystems should not be affected. *Iptable_filter* registers three hooks to netfilter and applies the iptable rules to network traffics at three guarding points of the netfilter: input, output and forward. We write a number of input and output iptable rules and neither of them actually blocks the network traffics during the test. Three benchmarks: bonnie [53], iperf [54] and lmbench [55], are performed to examine the performance of disk IO, network IO and the entire system respectively.

Comparing with VMM-only system (pure virtualization system), DARK's overhead comes from on-demand emulation, which is composed of two parts: 1. Context switch between virtualization and emulation; 2. Execution overhead in emulator, including binary translation, policy enforcement and execution of translated code sequences. To identify the contribution of each part to the overall cost, we devise another test system: DARK-CS. It does the context switch from virtualization to emulation when an *iptable_filter* function starts to run. Then, emulator returns the control back to VMM immediately and the *iptable_filter* function is actually executed over VMM. Therefore, context switch between virtualization and emulation is the only overhead

of DARK-CS. In the experiment, we run each benchmark in DARK, DARK-CS and VMM-only system.

Table 6. bonnie test result for 100 M files

	Sequential Output						Sequential Input				Random	
	Per Char		Block		Rewrite		Per Char		Block		Seeks	
	K/sec	%CPU	K/sec	%CPU	K/sec	%CPU	K/sec	%CPU	K/sec	%CPU	/sec	%CPU
VMM	8528±233	64.6±3	12755±1425	45.1±5	19082±1490	53.0±3	15805±2301	75.4±4	129292	71.3±2	3515±1908	84.7±4
DARK-CS	8038±345	61.5±5	11715±1379	41.2±6	17402±1834	48.2±2	16860±2004	80.3±5	130266	74.3±4	4969±1759	85.4±2
DARK	8168±405	67±6	13949±1106	43.7±5	18742±2046	49.8±2	14480±2720	73.4±7	125493	72.7±4	5117±1254	83.2±4

Table 6 shows the test result of bonnie. It's observed that three systems have little performance difference when running bonnie. This is because bonnie just accesses the files on disk and *iptable_filter* is not being executed. Bonnie's test result suggests that DARK's overall performance is same as VMM-only system when on-demand emulation doesn't take place.

Table 7. iperf test result for 30 seconds traffic

	VM as Server (M/sec)		VM as Client (M/sec)	
	TCP	UDP	TCP	UDP
VMM-only	21.8±1.2	1.05±0.1	26.8±2.3	1.13±0
DARK-CS	19.73±0.5	1.01±0	23.99±1.4	1.08±0.1
DARK	19.60±0.6	1.00±0.1	24.05±1.0	1.08±0.1

The iperf test result in table 7 reveals the impact of on-demand emulation on overall system performance. TCP and UDP throughputs of DARK-CS are slightly (about 10%) lower than VMM-only's, which indicates that the overhead of context switch is non-negligible but not significant. CPU state transferring, shadow page table synchronization and page fault handling are three main components of context switch in DARK. However, It is still unknown which component should take the responsibility of performance penalty at the moment. Further, it seems that neither component has much room left for performance improvement. Table 5 also suggests that DARK and DARK-CS have indistinguishable TCP and UDP throughputs. This result can be explained by the code caching technique introduced in section 5.2: to a block of module code, binary translation and policy enforcement are performed only at the first time this block of code is executed, and its translated code sequence in the code cache plays the primary role of deciding the performance in the long run. So code caching is effective to reduce the emulation overhead. We also did the performance test with the Imbench, and test result (Appendix A) confirmed the conclusions we draw above.

3.6 Discussion

To make DARK more applicable, it's desired to develop new approaches to further reduce the DARK's overhead. On strategy is to combine static analysis and dynamic analysis techniques to gain better understanding of a LKM's behavior. For example, one method could be not monitoring a callback function repeatedly for the same set of parameters if we can assure that they always lead to the same execution path. This can be done by examining the function's control flow graph (CFG) and data flow graph (DFG) that are constructed through static analysis. Another strategy is to tweak the tradeoff between performance and security: sacrificing a bit level of security to gain

better performance. One method for DARK could be to selectively switch to emulation based on certain conditions, e.g., current system load, virtual CPU usage and bandwidth usage. Apparently, this method achieves better performance at the cost of degrading the security level.

DARK's current policy database was manually built based on expert's knowledge, which could be challenged by those malware that explores vulnerabilities unknown to policy creators. So, it's necessary to seek approaches that automate the policy generation and increase the coverage of the policy base. One previous work that DARK can refer to is [40]. It uses the data mining technique to extract the rules that differentiate benign and malicious modules.

It should be pointed out that DARK is not designed to withstand the rookits that access to kernel in abnormal ways, e.g., directly writing kernel memory or injecting malicious code to kernel by exploiting the vulnerabilities of benign kernel code. These attacks have been well addressed by previous rootkit prevention systems [28] [38].

Last, DARK built its detector in the hypervisor layer in order to evade the attacks by malware in a guest OS. It largely depends on the assumption that hypervisor (emulator or VMM) is isolated from guest OS. In reality, this assumption may not be true because of the vulnerabilities in hypervisor software. Recent work [56] has found some vulnerabilities in the main hypervisor software that can be explored to penetrate the hypervisor layer. This problem outreaches the scope of DARK, and other security mechanisms have to be relied on to address it.

CHAPTER 4

KERNEL MALWARE ANALYSIS

In the chapter, we present a proof-of-concept system, Rkprofiler, in attempt to provide comprehensive profile of kernel malware. Rkprofiler is built based on the PC emulator QEMU and analyzes Windows rootkits. The binary translation of QEMU allows Rkprofiler to sandbox rootkits and inspect each executed malicious instruction. Further, Rkprofiler develops the memory tagging technique to perform just-in-time symbol resolving for memory addresses visited by rootkits. Combining deep inspection with the memory tagging, Rkprofiler is able to track all function calls and most kernel object accesses made by rootkits.

4.1 Motivation

When an attacker breaks into a machine and acquires administrator privileges, kernel malware could be installed to serve various attacking purposes (e.g., process hiding, keystroke logging). The complexity of attackers' activity on machines has significantly increased. Rootkits now cooperate with other malware to accomplish complicated tasks. For example, the rootkit Rustock.B has an encrypted spam component attached to its code image in memory. The initialization routine of this rootkit registers a notification routine to the Windows kernel by calling the kernel function PsCreateProcessNotifyRoutine. This notification routine is then invoked each time that a new process is created. When detecting the creation of Windows system process Service.exe, Rustock.B decrypts the spam components and injects two threads into the Service.exe process to execute the spam components [57]. Without understanding the behavior of the Rustock.B rootkit, it would be difficult to determine how the spam threads are injected into the Service.exe process. To fully comprehend

malicious activities on a compromised machine, it is necessary to catch and dissect key malware that attackers have loaded onto the machine. Thus, analyzing rootkits is an inevitable task for security professionals.

Most of the early rootkits were rudimentary in nature and tended to be single-mission, small and did not employ anti-reverse engineering techniques (e.g., obfuscation). These rootkits could be manually analyzed using disassemblers and debuggers. Since rootkit technology is much more mature today, the situation has changed. Rootkits have more capabilities and their code has become larger and more complex. In addition, attackers apply anti-reverse engineering techniques to rootkits in order to prevent people from determining their behavior. Rustock.C is one such example. The security company, Dr. Web, who claimed to be one of the pioneers that provided defense against Rustock.C, took several weeks to unpack and analyze the rootkit [58]. The botnet using Rustock.C was the third largest spam distributor at that time, sending about 30 million spam messages each day. This example illustrates how the cost incurred by the delay of analyzing kernel malware can be huge. As another example, the conficker worm that has infected millions of machines connected to the Internet was reported by several Internet sources [59] [60] (on April 8th 2009) that a heavily encrypted rootkit, probably a keylogger, was downloaded to the victim machines. At the time of the writing of this dissertation, which was three days later, no one had published the details of the rootkit. It is still unclear how severe the damage (e.g., economic, physical) will be as a result of this un-dissected rootkit. Accordingly, developing new approaches for quickly analyzing rootkits is urgent and also critical to defeating most rootkit-involved attacks.

Several approaches have been proposed to address the rootkits analysis problem to some extent. For examples, HookFinder [4] and HookMap [5] are two rootkit hooking

detection systems. The former uses dynamic data tainting to detect the execution of hooked malicious code; and the latter applies backward data slicing to locate all potential memory addresses that can be exploited by rootkits to implant hooks. K-tracer [6] is another rootkit analysis system that uses data slicing and chopping to explore the sensitive kernel data manipulation by rootkits. Unfortunately, these systems cannot meet the goal of comprehensively revealing rootkit behavior in a compromised system. Meeting this goal requires answering two fundamental questions: 1) what kernel functions have been called by rootkits? and 2) what kernel data objects have been visited by rootkits? Rkprofiler is such system that aims to provide the answers to these two questions.

4.2 Challenges

Modern operating systems (OSs) like Windows and Linux utilize two ring levels (ring 0 and 3) provided by X86 hardware to establish the security boundary between the OS and applications. Kernel instructions and application instructions run at ring level 0 and 3 respectively (also called kernel mode and user mode). The execution of special system instructions (INT, SYSENTER and SYSEXIT) allows the CPU to switch between kernel mode and user mode. This isolation mechanism guarantees that applications can only communicate with the kernel through well-defined interfaces (system calls) that are provided by the OS. Many sandbox-based program analysis systems take advantage of this isolation boundary and monitor the system calls made by malware [61] [62]. While this approach is effective to address user-space malware, it fails to address kernel malware. This is because there is no well-defined boundary between benign kernel code and malicious kernel code. Kernel malware possess the highest privileges and can directly read and write any kernel objects and system resource. Moreover, kernel malware may have no constant ``identity" - that is, some

kernel malware could be drivers and others could be patches to benign kernel software. So the first challenge is how to create a ``virtual" boundary between kernel malware and benign kernel software. Rkprofiler overcomes this challenge by using the timing characteristic of malware analysis. Before loading kernel malware, all kernel code is treated as benign code; after loading kernel malware, newly loaded kernel code is considered malicious. Note this ``virtual" boundary only isolates code, but not data. This is because the data created by malicious code can also be accessed by benign code, and Rkprofiler does not monitor the operations of benign kernel code for the purpose of design simplicity and better performance.

When monitoring a VM at the hypervisor layer, only hardware-level activities (e.g., memory reads and writes) are observed. To make these observations useful, it is necessary to translate the hardware-level activities to software-level activities. Here, software-level activities refer to using software terms to describe program activities. For example, local variable X is modified. This translation requirement is also known as the semantic gap problem [18]. This problem can be expressed as the following: given a memory address, what is its symbol? Automatically finding the symbols for static kernel objects (global variables and functions) is straightforward, but automatically finding the symbols for dynamic kernel objects (data on stack and heap) is challenging. This challenge is not well addressed by previous work. In this research, we propose a method called *aggressive memory tagging* (AMT) to overcome this challenge. The basic idea of AMT is to perform the symbol resolution at run time and derive the symbols of dynamic kernel objects from other kernel objects whose symbols have been identified. It should be pointed out that Microsoft does not publish all kernel symbols and we can only gather the kernel symbols that are publically available (Microsoft symbol server, DDK documents and some

unofficial Internet sources). So the current implementation of Rkprofiler is not able to resolve many unpublished symbols. Nevertheless, we find that it identifies most sensitive available symbols in our evaluation.

4.3 System Description

Rkprofiler is composed of four software components: *generator*, *controller*, *monitor* and *reporter*. These software components operate in three phases temporally: *pre-analysis*, *analysis* and *post-analysis*. In the pre-analysis phase, the generator collects symbols of native Windows kernel modules (e.g., ntoskrnl.exe, ndis.sys) from the program database (PDB) files available on the Microsoft symbol server [63] and header files in Microsoft's Driver Development Kit (DDK). Two databases are produced by the generator at the end of this stage: type graph and system map. The type graph database contains the data type definitions of native Windows kernel modules. There are six classes of data types: basic type, enum, structure, function, union, and pointer. The data types in the last four classes are considered as composite data types, indicating that a data type includes at least one sub data type. For example, the sub data types of a structure are data types of its data members. In the type graph database, Rkprofiler assigns a unique type ID to each data type. A data type is represented by its type ID, type name, size, class ID and class specific data (e.g., the number of sub data types and their type IDs). The system map database keeps the names, relative virtual addresses and type IDs of global variables and functions used by native Windows kernel modules. In addition, the names and type ID of parameters and the return value for each function are also stored in system map. The generator is comprised of several executables and Perl scripts.

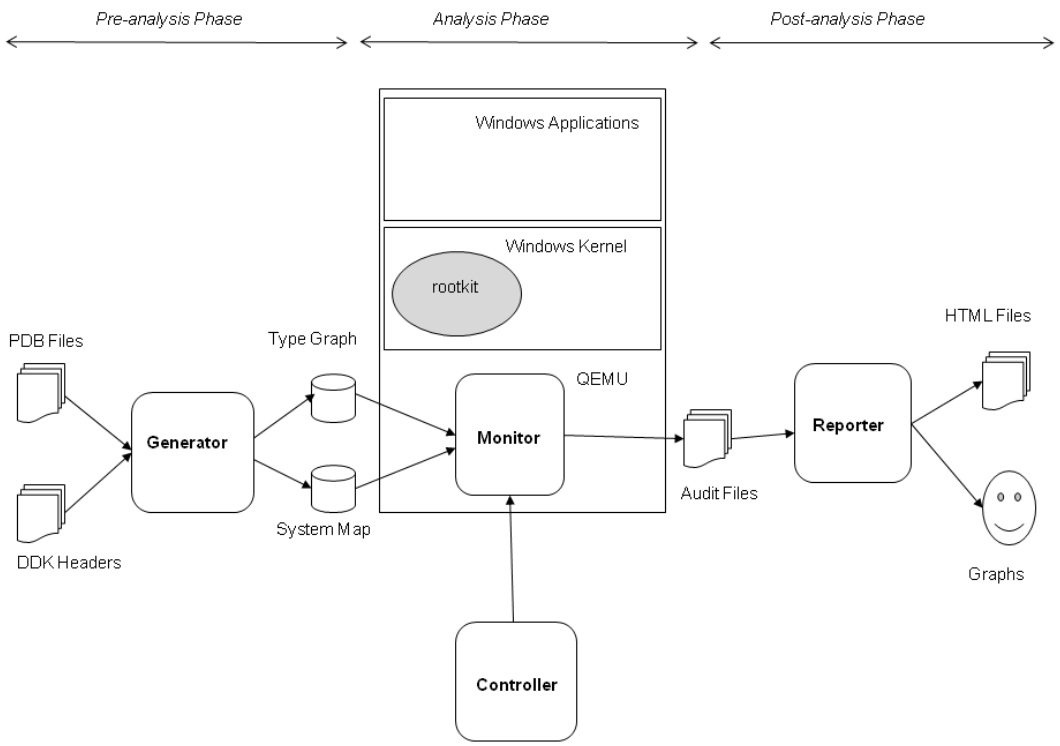


Fig. 5. Rkprofiler architecture and rootkit analysis process

Executing malware and monitoring its behavior are carried out in the analysis phase. Two components of Rkprofiler, controller and monitor, are involved in this phase. The monitor is built into QEMU. The controller is a standalone shell script that sends commands to the monitor via the Linux signal mechanism. Four commands are defined in their communication messages: RKP_INIT, RKP_RUN, RKP_STOP, and RKP_REPORT,, (which are explained shortly). First, a test VM is started and goes into a clean state in which no malware is installed and executed. Then, the controller sends a RKP_INIT command to the monitor. After receiving the command, the monitor queries the kernel memory image of the guest OS and creates a hash table of trusted kernel code. Next, the controller instructs the monitor to start monitoring through a RKP_START command. At that point, Rkprofiler is ready for the monitoring task. For example, a user starts executing malware in the VM. Depending

on the attack objectives of the malware, the user may run other applications to trigger more behaviors from the malware. For example, if the malware is intended to hide processes, the user may open the Windows task manager to induce the hiding behavior. Since the malware can be tested repeatedly, the attack objectives of the malware can be inferred from the analysis results of previous tests. To obtain the monitoring result or end the test, the user can have the controller issue RKP_REPORT or RKP_STOP commands to the monitor. The first command informs the monitor to write the monitoring result to local audit files; the second command prompts the monitor to stop monitoring and clear its internal data structures. Four audit files in CSV format are generated in the analysis phase: trace, tag trace, tag access trace, and system resource access trace. These files contain the functions called by the malware, their parameters and return values, kernel data objects visited by the malware and their values. In the post-analysis phase, the reporter is executed to create user-friendly reports. Using the audit files generated in the analysis phase, the reporter performs three tasks. First it builds a call graph from the call trace and saves the graph to another file; second, it visualizes the call graph and tag trace with open-source software GraphViz [64]; third it generates the HTML-formatted reports for call traces and tag traces (CSV format). The entire analysis process is illustrated in Figure 5.

The monitor component of Rkprofiler was built based on the open-source PC emulator QEMU. To support multiple CPU architectures, QEMU defines an intermediate instruction set. When QEMU is running, each instruction of a VM is translated to the intermediate instructions. Rkprofiler performs code inspection and analysis at the code translation stage. To improve the performance, QEMU caches the translated Translation Block (TB) so that it can be re-executed on the host CPU over time. However, this optimization approach is not desirable to Rkprofiler because an

instruction can behave differently in varied machine states. For example, the instruction CALL, whose operand is a general-purpose register, may jump to diverse instructions depending on the value of that register. For each malicious TB that has been cached, Rkprofiler forces QEMU to always perform the code translation. But, the newly generated code is not stored in the cache and the existing cached code is actually executed. Another problem arises when a TB contains multiple instructions. In QEMU, VM states (register and memory contents) are not updated during the TB translation. Except for the first instruction, the translation of all other instructions in a TB could be accompanied by incorrect VM states, possibly resulting in analysis errors. Rkprofiler addresses this problem by making each malicious TB include only one instruction and disabling the direct block chaining for all malicious TBs.

4.4 Malicious Code Detection

Kernel malware could take the form of drivers and be legitimately loaded into the kernel. They can also be injected into the kernel by exploiting vulnerabilities of benign kernel software. Rkprofiler is designed to detect kernel malware that enter the kernel in both ways. Roughly speaking, before any malware is executed, Rkprofiler looks up the kernel memory image and identifies all benign kernel code in the VM. Then it groups them into a Trust Code Zone (TCZ) and a hash table is created to store the code addresses of the TCZ. When malware is started, any kernel code that does not belong to the TCZ is regarded as malicious and therefore is tracked by Rkprofiler.

Identification of the trusted kernel code is straightforward if the non-execute (NX) bit of the page table is supported by the (virtual) Memory Management Unit (MMU) of a (virtual) processor. In this case, the kernel code and data do not co-exist in any page of memory. Rkprofiler just needs to traverse the page table of a process to find out all the executable kernel pages. QEMU can provide a NX-bit enabled virtual

processor (by enabling the PAE paging mechanism), but this system configuration is not common. Doing so may influence the malware behavior in an undesired manner. For example, the malware could stop running when it detects that the (virtual) CPU is NX enabled. So, the current implementation of Rkprofiler does not require enabling the NX-bit of the virtual CPU. Instead, it interprets all images of benign kernel modules and obtains the Relative Virtual Addresses (RVA) of the code sections. Then it computes their actual virtual addresses by adding the RVAs to the module base addresses, which is acquired by scanning the kernel memory of the VM. After that, Rkprofiler stores the TCZ addresses in a hash table. However, one common type of kernel malware attack is to patch the benign kernel code. To accommodate this type of attack, Rkprofiler excludes the patched code from the TCZ and revises the TCZ hash table at run time. Rkprofiler identifies the patched code by examining memory write operations and memory copy functions that the malware performs. Note, malware could escape this detection by indirectly modifying the TCZ code (e.g., tampering with kernel memory from user space). A more reliable method is to monitor the integrity of the TCZ as [33] does. Last, Rkprofiler determines whether a kernel TB is malicious or not right before it is translated. If the address of a TB is not within the TCZ, it is deemed as a malicious TB. The hash table implementation of the TCZ ensures that malicious code detection has a small performance hit on the entire system.

4.5 Function Tracking

Kernel malware often interacts with the rest of the kernel by calling functions exported by other kernel modules. In Rkprofiler, we use the terms I2E (Internal-to-External) and E2I (External-to-Internal) to describe the function-level control flow transferring between malicious code and benign code. Here, internal and external

functions refer to the malicious function code and benign function code respectively. Function calls and returns are two types of events that Rkprofiler monitors. For example, *I2E call* indicates the event that an internal function invokes an external function; *I2E return* refers to the event that an internal function returns to its caller that is an external function. Capturing these function events is important for Rkprofiler to reveal the activity of the malware. Further, in an instance, the kernel malware may directly call the registry functions exported by ntoskrnl.exe like zwSetKeyValue to manipulate local registry entries. Rkprofiler is also designed to capture the I2I (Internal-to-Internal) call and return events. By doing so, Rkprofiler is able to construct (partial) call graphs of the kernel malware, which helps a security professional understand the code structure of the malware. This capability is important, especially when the malware is obfuscated to resist static code analysis. Note, E2E (External-to-External) function events are not monitored here because Rkprofiler does not inspect benign kernel code.

To completely monitor the function-level activity of malware, a data structure called *function descriptor* is defined to represent a stack frame (activation record) of a kernel call stack, allowing Rkprofiler to track the call stacks of the kernel malware. When a function that is called by malware is detected, Rkprofiler creates a new function descriptor object and pushes it to the stack. Conversely, when the function is returned, its function descriptor object is popped from the stack and is deleted. One function descriptor has a pointer that points to the function descriptor of the caller. This pointer is used by Rkprofiler to construct the caller-callee relationships in the post-analysis phase.

4.5.1 Function Call and Return Detection

The method of detecting a function call event depends on the calling directions. For I2I and I2E calls, Rkprofiler monitors the CALL instructions executed by the malware. Further, it can obtain the function address from the operand of a CALL instruction and the return address that is next to the CALL instruction. For E2I calls, a CALL instruction belongs to TCZ and is not monitored by Rkprofiler. So, the detection point is moved to the first instruction of the callee function. To capture E2I calls, Rkprofiler adds extra data members to the TB descriptor *TranslationBlock*. The first data member indicates what the last instruction of this TB is: CALL, JMP, RET or others. If it is a CALL instruction, the second data member records the return address of the call. Rkprofiler fills in the two data members of a TB when it is being translated. In addition, Rkprofiler creates a global pointer that points to the last TB descriptor whose code was just executed by the virtual CPU. Before translating a malicious TB, Rkprofiler queries the last TB descriptor to decide if it is an E2I call event. The decision is based on three criteria: 1) if the last TB is benign; 2) if the last instruction of the last TB is CALL; and 3) if the return address stored in the kernel stack is equal to the one stored in the last TB descriptor. The reason for criterion 3 is that the return address is always constant for both direct and indirect calls. On the other hand, Rkprofiler processes the function return events in a similar way to the call events: for I2I and E2I returns, Rkprofiler captures these events by directly monitoring the RET instructions executed by the malware; for I2E returns, Rkprofiler detects them at the instructions directly following the RET instructions and the criteria of the decision are similar to that for the E2I calls.

Two problems complicate the call event detection methods described above. The first one is a *pseudo function call*, which is caused by JMP instructions. When a kernel module attempts to invoke one function exported by another kernel module, it

first executes the CALL instruction to invoke an internal stub function and the stub function then jumps to the external function by running the JMP instruction. Normally, the internal stub function is automatically generated by a compiler and the operand of the JMP function is an IAT entry of this module, whose value is determined and inserted by the system loader. Without recognition of these JMP instructions, Rkprofiler incorrectly treats an I2E call as an I2I call: labeling the new function descriptor with the internal stub function address. One example of such functions is DbgPrint. To address a pseudo function call, Rkprofiler first creates an I2I function descriptor and labels it with the internal stub function address. When detecting if an internal JMP instruction is executed in order to jump to an external address, Rkprofiler locates the I2I function descriptor from the top of the function tracking stack, and replaces the internal address with the external address. The second problem is an *interrupt gap*. This is where an interrupt is sent to the (virtual) CPU while it is executing an E2I CALL (or I2E RET) instruction. Consequently, some interrupt handling instructions are executed between the E2I CALL (or I2E RET) instruction and the subsequent internal instruction that Rkprofiler monitors. In this situation, the last TB descriptor does not record the expected CALL (or RET) instruction, so Rkprofiler is unable to track the E2I call (or I2E return) event and observes an unpaired return-call event. The solution to this problem is part of our future work. Fortunately, we did not see interrupt gaps in the experiments.

4.5.2 System Context

As an extension to the kernel software, a kernel malware could run in multiple system contexts. Under some circumstances, finding the running context of a malware can benefit the user's understanding of its behavior. In an instance, when malware code runs in an interrupt context, it means that the malware probably has

compromised interrupt handling routines. The relevant interrupt request (IRQ) numbers can provide more information about the attacks, e.g., manipulating IRQ 1 triggered by keyboard almost implies a key logger at once. In Rkprofiler, Qemu helps determining interrupt contexts. Specifically, whenever a CPU receives an interrupt, it need conduct a series of operations to preserve the current system context and start the relevant interrupt handling routine, e.g., pushing the EIP register to stack. When the interrupt routine returns, the IRET instruction is executed to restore the previous system context. Correspondingly, Qemu need simulate the CPU's handling of an interrupt at the two moments: CPU’s entering the interrupt context and CPU’s leaving the interrupt context. Rkprofiler places a global variable in the interrupt simulation code of Qemu, and it can indicate if a VM enters or leaves an interrupt context. For process contexts, Pkprofiler reads the relevant process information of the current process from its process descriptor object (EPROCESS) in the kernel, e.g., process id and image name. These data are stored in a data structure called *context descriptor*. Rkprofiler associates each monitored function with one *context descriptor* object, representing the actual running context of this process. Note only E2I calls require Rkproifler querys system context each time, and the context information of I2E and I2I calls can be inherited from of their callers.

4.5.3 Parameter Tracing

To track the parameters of an interest function, Rkprofiler first checks if this function is semantically identifiable: an external function in the system map or a tagged function whose type definition can be found in the type graph. Section 4.6 discusses the details of how functions are tagged. Rkprofiler traces a function parameter with three attributes: type id, name and IO flag. The IO flag indicates whether this parameter is an input or output of the function. Name and IO indicator

may not be available for every function, so they are optional attributes. Here, the return value of a function is treated as one output parameters. When monitoring an interest function, Rkprofiler records the input parameters before it starts, and records the output parameters after it returns. In default, one parameter is processed as an input parameter.

To locate function parameters in a VM, it is necessary to understand the calling conventions of Windows operating system. They influence the ways that parameters are passed to functions. Windows adopts three calling conventions: _cdecl, _stdcall and _fastcall. _cdecl is the default calling conventions for C and C++ functions. Under this convention, a caller can pass various numbers of parameters to its callee and it is responsible for cleaning the parameters in the stack after the callee returns. Most of Windows kernel C Run-Time (CRT) functions follow this calling convention. A _stdcall function has a constant number of parameters, which allows compiler to calculate total parameter size and make the callee function clean the stack. So _stdcall usually generates smaller code size than _cdecl. Most Windows kernel APIs use _stdcall. When calling a _cdecl or _stdcall function, CPU pushes the function parameters to stack in the reversing order (from right to left). The _fastcall calling conversion puts the first two parameters to ECX and EDX registers and the rest parameters to stack in the reversing order. The return value of a function is always passed through the EAX register regardless of its calling convention. Rkprofiler finds the calling convention of a function (or function type) from the system map or type graph.

```
NTSTATUS
ZwOpenKey(
    OUT PHANDLE             KeyHandle,
     IN ACCESS_MASK         DesiredAccess,
    IN POBJECT_ATTRIBUTES   ObjectAttributes
);
```

Besides calling conventions, Rkprofiler also need have the knowledge of parameter sizes and stack layout. Parameter sizes are stored in type graph, and stack layout is a prior knowledge as described in [65]. The following example illustrates how Rkprofiler tracks function parameters. The Windows kernel function ZwOpenKey can be called by malware to open an existing registry key. The declaration of this function is shown in graph XX. ZwOpenKey has one output parameter and two input parameters. When a malware executes the CALL instruction to invoke this function, Rkprofiler captures the event and starts acquiring the parameters as below. Rkprofiler first queries the system map to obtain th ZwOpenKey e parameter lists and their type ids. Then, it realizes that two input parameters *DesiredAccess* and *ObjectAttributes* should be gathered at this moment. Because ZwOpenKey is a _stdcall function, all three parameters are stored in the stack. Rkprofiler also knows that parameters are pushed to the stack from the right to left, and the stack pointer (ESP) is pointing to the first parameter *KeyHandle*. Thus, Rkprofiler can figure out the addresses of *DesiredAccess* and *ObjectAttributes* and read their values. After ZwOpenKey executes the RET instruction, Rkprofiler detects the event and starts reading two output parameters. Although the parameter *KeyHandle* is literally popped up and the ESP is pointing to the item below the parameters in the stack at this point, the parameter value is still stored in the stack. Rkprofiler can compute the address of KeyHandle and read its value. Finally, the return value, whose type is NTSTATUS, is obtained from the EAX register.

4.6 Memory Tagging

Rkprofiler observes the hardware-level activities of kernel malware, so they should be translated to software-level activities to be understandable to users. Thus, given a virtual address that the malware visits, Rkprofiler is required to find its symbols (e.g.,

variable name and type). Here, we name the process of finding symbols for kernel objects as memory tagging. A memory tag is composed of tag ID, virtual address, type ID, variable name (optional) and parent tag ID (optional). If a kernel object is owned by the malware, it is an internal kernel object; otherwise, it is an external kernel object. If a kernel object is located in the dynamic memory area (stack and heap), it is a dynamic kernel object; otherwise, it is a static kernel object. Rkprofiler tags four types of kernel objects: static internal, dynamic internal, static external and dynamic external. Static external kernel objects include global variables and Windows kernel functions. Their symbols are stored in a system map. Tagging a static kernel object is straightforward. Rkprofiler searches the system map by its virtual address and the hit entry contains the target symbols. However, tagging a dynamic kernel object is challenging because its memory is dynamically allocated at run time and the memory address cannot be predicted. Attackers often strip off the symbols of their malware in order to delay reverse engineering, so Rkprofiler assumes that malware samples do not contain valid symbols.

Previous Linux rootkit detection systems [27] [31] present one approach of tracking dynamic kernel objects. A rootkit detector first generates a kernel type graph and identifies a group of global kernel variables. At run time, it periodically retrieves the dynamic objects from the global variables based on the graph type. For example, if a global variable is a linked list head, the detector traverses the list under the direction of the data structure type of list elements. Unfortunately, this approach cannot be applied to the task of profiling kernel malware. First, it covers a limited number of kernel objects, and many other kernel objects such as functions and local variables are not included. Second, since the creation and deletion of dynamic kernel objects could occur at any time, the time gap between every two searches in this approach will

produce inaccurate monitoring results. Last, this approach may track many kernel objects that the malware never visits. In this research work, we propose a new symbol exploration approach, *Aggressive Memory Tagging* (AMT), that can precisely find symbols for all kinds of static and dynamic kernel objects at a low computation cost.

4.6.1 AMT Description

We define a kernel object as *contagious* if another kernel object can be derived from it. *Tag inferring* is a process where a kernel object (child object) is derived from another (parent object). Two types of kernel objects are considered contagious: pointers and functions. A pointer kernel object could be a pointer variable or a structure variable containing a pointer member. The child object of a pointer is the pointee object. For a function, its child objects are the parameters and return value of this function. AMT follows the principle of the object tracking approach described above: tracing the dynamic objects from the static objects. Specifically, Rkprofiler first tags all static kernel objects that the malware accesses (memory reads/writes and function calls) by querying the system map. Then, the child objects of the existing contagious tags are tagged via tag inferring. This process is repeated until the malware stops execution or the user terminates monitoring. Note, a tag could become invalid in two scenarios: 1) if when a function returns, the tags of its local variables are invalidated; and 2) if a memory buffer is released, the associated tag becomes out of date as well. Only valid tags can generate valid child tags.

Rkprofiler performs tag inferring through a pointer object at the time that the malware reads or writes the pointer object. The reason is as follows: when reading a pointer, the malware is likely to visit the pointee object through the pointer; when writing a pointer, the malware will possibly modify the pointer to point to another object if the new value is a valid memory address. Because the executions of benign

kernel code are not monitored by Rkprofiler, both read and write operations over a pointer have to be tracked here. If only read operations are monitored, Rkprofiler cannot identify the kernel objects whose pointers are written by malicious code and read by benign code. Many hooks implanted by rootkits fall into this scenario. Similarly, if only write operations are monitored, Rkprofiler can miss the reorganization of kernel objects whose pointers are written by benign code and read by malicious code. Many external kernel objects that are visited by rootkits fall into this scenario. The procedure of tag inferring through a pointer object is as follows: 1) Rkprofiler detects a memory read or write operation and searches the tag queue to check if the target memory corresponds to a contagious tag; 2) if yes, Rkprofiler obtains the up-to-date pointer value and verifies that it is a valid memory address; 3) Rkprofiler searches the tag queue to check if the pointee object is tagged; 4) if not, Rkprofiler obtains the symbols of the pointee object from the type graph and creates a new tag. On the other hand, when a recognizable function is called, tag inferring through the function object is carried out by identifying the function parameters. Input parameters are tagged when the function is called; output parameters are tagged when the function returns.

4.6.2 Implementation

Rkprofiler creates a data structure called *tag descriptor* to represent memory tags. A tag descriptor includes the virtual address of the tag, type ID, a boolean variable, a num variable for memory type, one pointer to the parent tag and one pointer to the function descriptor. The Boolean variable indicates if a tag is contagious or not. The memory type member tells if the tagged object is on the stack, heap or another memory object. Rkprofiler monitors the kernel memory management functions called by malware and records it to a heap list (the memory buffers allocated to the

malware). When a buffer is released, Rkprofiler removes it from the heap list. The function descriptor member of a tag helps identify which function is running when this tag is generated. Finally, Rkprofiler maintains a tag queue that contains all the tags that have been created. When a tag is created, its tag descriptor is inserted into the tag queue. The tag is removed from the tag queue after it becomes invalid. Because malware's memory accesses are frequent events, Rkprofiler needs to search the tag queue frequently as well. The tag queue describes a group of various-sized memory segments. If it is organized as a list structure like a linked list, its linear searching time is expensive. To address the problem, Rkprofiler applies the approach presented in Chapter 3 that converts a group of various-sized memory segments to a hash table. The basic idea is to break a memory segment into a number of fix-sized memory segments (buckets). A list structure is stored in one bucket to handle the case that some portions of the bucket should not be counted. In this way, the time for searching the tag queue becomes constant.

The Windows kernel provides built-in supports for linked lists via two data structures: SINGLE_LIST_ENTRY (for single linked list) and LIST_ENTRY (for double linked list). Several kernel APIs are available to simplify driver developers' tasks when managing linked lists (e.g., adding or removing elements). However, this support causes problems to the memory tagging process of Rkprofiler. For example, in a double linked list, each element contains a data member whose data type is LIST_ENTRY. Two pointers of this data member point to the LIST_ENTRY data members of two neighbor elements. When one list element is tagged and malware tries to visit the next list element from this one, Rkprofiler just tags the LIST_ENTRY data member of the next list element with the type LIST_ENTRY. This is not acceptable because what Rkprofiler wants to tag is the next list element with its type.

In the pre-analysis stage, we annotated the SINGLE_LIST_ENTRY and LIST_ENTRY data members with the type names of list elements and their offsets. When parsing the type header file, the generator replaces the SINGLE_LIST_ENTRY and LIST_ENTRY data members with pointers to list elements. The offset values are also stored in the type graph, allowing the monitor to find the actual addresses of neighbor elements. Another problem is relative pointers. The Windows kernel sometimes uses relative pointers to traverse a list in the following way: the address of the next element is computed by adding the relative pointer and the address of the current element. One example is the data buffer that contains the disk file query result by kernel function NtQueryDirectoryFile. Because these relative pointers are defined as unsigned integer, we also need to label the relative pointers in the kernel type header file such that Rkprofiler can recognize them and properly compute the element addresses.

Rkprofiler has to handle two ambiguous data types that the Windows kernel source uses. The first one is *union*. Union is a data type that contains only one of several alternative members at any given time, and the memory storage required for a union is decided by its largest data member. Unfortunately, guessing which data member of a union should be used at a given time depends on code context, which is hard to automate in Rkprofiler. The second one is generic pointer *pvoid*. Pvoid can be caste to another data type by developers. The actual data type that pvoid points to at a given time is context dependent too. Automatically predicting the pointee data type for pvoid is another challenge. The current default solution is to replace a union with one of its largest members and leave pvoid alone. While performing the analysis, a user can modify the kernel data type header file and change the definition of union or

pvoid in terms of his understanding of their running contexts. An automated solution has been proposed in [72].

4.7 Hardware Access Monitoring

In comparison to user-space malware, kernel malware is able to bypass the mediation of the OS and directly access low-level hardware resources. In X86 architectures, in addition to the memory and general-purpose registers that kernel malware access through instructions like MOV and LEA, other types of system storage resources could also be visited and manipulated by kernel malware. CPU caches (e.g., TLB) dedicate registers and buffers of I/O controllers. Attackers have developed techniques that take advantage of these hardware resources to devise new attacks. For example, upon a system service (system call) invocation made by a user-space process, Windows XP uses instruction SYSENTER (for Intel processor) to perform the fast transition from user space to kernel space. The entry point of kernel code (a stub function) is stored in a dedicated register called IA32_SYSENTER_EIP, which is one of Model-Specific Registers (MSRs). When executing SYSENTER, the CPU sets the EIP register with the value of IA32_SYSENTER_EIP. Then, the kernel stub function is called and it transfers the control to the target system service. To compromise Windows system services, a rootkit could alter the system control-flow path by resetting the IA32_SYSENTER_EIP to the starting address of a malicious stub function, and this function can invoke a malicious system service. So, capturing the malware's accesses to these sensitive hardware resources could be essential to comprehend its attacking behavior. Currently, Rkprofiler monitors twenty system instructions that malware might execute. They are not meant to be complete at this point and can be expanded in the future if necessary.

4.8 Case Studies

4.8.1 FUTo

FUTo is an enhanced version of the Windows kernel rootkit FU, which uses the technique called Direct Kernel Object Manipulation (DKOM) to hide processes and drivers and change the process privileges. DKOM allows rootkits to directly manipulate kernel objects, avoiding the use of kernel hooks to intercept events that access these kernel objects. For example, a rootkit can delete an item from the MODULE_ENTRY list to hide a device driver without affecting the execution of the system. This technique has been applied to many rootkit attacks, such as hiding processes, drivers and communication ports, elevating privilege levels of threads or processes and skewing forensics [47] In this experiment, FUTo was downloaded from [66] and it included one driver (msdirectx.sys) and one executable (fu.exe). The fu.exe was a command-line application that installed the driver and sent commands to the driver according to the user's instructions. During the test, we executed the fu.exe to accomplish the following tasks: querying the command options, hiding the driver (msdirect.sys) and hiding the process (cmd.exe). After that, we used Windows native system utilities (task manager and driverquery) to verify that the target driver and process did not show up in their reports. The test took less than 3 minutes.

Table 8. FUTo tag trace table

Tag ID	Address	Type	Parent Tag	Category	Size (bytes)	Process ID	Process Name
0	0xf6b7e7e6	FYBCT_0049_0953_DriverInit	n/a	function	n/a	4	System
1	0x825c3978	DRIVER_OBJECT	n/a	struct	168	4	System
2	0x827cba00	EPROCESS	n/a	struct	608	4	System
3	0x825991c8	EPROCESS	2	struct	608	4	System
4	0x825ce020	EPROCESS	3	struct	608	4	System

5	0xf7b03c58	PVOID	n/a	pointer	4	4	System
6	0xf6b9b92	PDEVICE_OBJECT	n/a	pointer	4	4	System
7	0xf6b7e722	FUNCT_0049_095B_MajorFunction	1	function	n/a	4	System
8	0xf6b7d43a	FUNCTION_00BC_0957_DriverUnload	1	function	n/a	4	System
9	0x82604600	MODULE_ENTRY	1	struct	52	4	System
10	0x82609f18	DEVICE_OBJECT	n/a	struct	184	1920	Fu.exe
11	0x8266fc28	IRP	n/a	struct	112	1920	Fu.exe
12	0x8266fc03	IRP	n/a	struct	112	1920	Fu.exe
13	0x826bc118	IRP	n/a	struct	112	1952	Fu.exe
14	0x826bc103	IRP	n/a	struct	112	1952	Fu.exe
15	0x826d8288	MODULE_ENTRY	9	struct	52	1952	Fu.exe
16	0x8055ab20	MODULE_ENTRY	9	struct	52	1952	Fu.exe
17	0x826bc210	IRP	n/a	struct	112	1952	Fu.exe
18	0x825d1020	EPROCESS	4	struct	608	1880	Fu.exe
19	0x8273a7c8	EPROCESS	18	struct	608	1880	Fu.exe
20	0x826eb408	EPROCESS	19	struct	608	1880	Fu.exe
21	0x825d5a800	EPRCOESS	20	struct	608	1880	Fu.exe
22	0x825e4da0	EPRCOESS	21	struct	608	1880	Fu.exe
23	0x825a9668	EPRCOESS	22	struct	608	1880	Fu.exe
24	0x82695180	EPRCOESS	23	struct	608	1880	Fu.exe
25	0x825a0da0	EPRCOESS	24	struct	608	1880	Fu.exe
26	0x82722980	EPRCOESS	25	struct	608	1880	Fu.exe
27	0x825c27e0	EPRCOESS	26	struct	608	1880	Fu.exe
28	0x82624bb8	EPRCOESS	27	struct	608	1880	Fu.exe
29	0x825de980	EPRCOESS	28	struct	608	1880	Fu.exe
30	0x8248bda0	EPRCOESS	29	struct	608	1880	Fu.exe
31	0x8264a928	EPRCOESS	30	struct	608	1880	Fu.exe
32	0x8263a5a8	EPRCOESS	31	struct	608	1880	Fu.exe
33	0x825d9020	EPRCOESS	32	struct	608	1880	Fu.exe
34	0xe13ed7b0	HANDLE_TABLE	4	struct	68	1880	Fu.exe
35	0x82607d48	ETHREAD	32	struct	600	1880	Fu.exe
36	0xe15ca640	HANDLE_TABLE	32	struct	68	1880	Fu.exe
37	0xe10a8a08	HANDLE_TABLE	36	struct	68	1880	Fu.exe
38	0xe1747cd0	HANDLE_TABLE	36	struct	68	1880	Fu.exe

We compared the call graph created by Rkprofiler with the call graph created by IDA-pro (which uses the static code analysis technique). It was found that the former was the sub-graph of the latter, which is as expected. The tag trace table of the Futo is shown in Table 8 (Appendix B is the corresponding tag trace graph). The driver msdirectx was executed in four process contexts in the graph. Process 4 (System) is the Windows native process that was responsible for loading the driver misdirectx. The driver initialization routine (with tag ID 0) was executed in this process context. The other three processes were associated with FUTo.exe and they communicated with the misdirectx driver to perform the tasks of hiding the driver and process. One important observation is that the major attacking activities have been recorded by Rkprofiler and can be easily identified in the tag trace table by users. To hide itself, the driver msdirectx first reads the address of its module descriptor (with tag ID 9) from its driver object (with tag ID 1). Then it removes this module descriptor from the kernel MODULE_ENTRY list by modifying the Flink and Blink pointers in two neighbor module descriptors (tag ID 15 and 16). Similarly, to conceal process cmd.exe, msdirectx first obtains the process descriptor (with tag ID 2) of the current process by calling kernel function IoGetCurrentProcess. Starting from this process descriptor, msdirectx traverses the kernel EPROCESS list to find the process descriptor (with tag ID 4) of process csrss.exe. These two steps take place in the System process context. After receiving the command for hiding the cmd.exe process sent by one of the fu.exe processes, msdiretx searches the kernel EPROCESS list, beginning with the process descriptor of csrss.exe. When the process descriptor (with tag ID 32) of cmd.exe is found, msdirectx removes it from the kernel EPROCESS list by altering Flink and Blink pointers in two neighbor process descriptors (with tag ID 31 and 33). Furthermore, Flink and Blink pointers in the process descriptor of

cmd.exe are also modified to prevent the random Blue Screen of Death (BSOD) when exiting the hidden process. To evade the detection of rootkit detectors, FUTo deletes the hidden process from the other three kernel structures: kernel handle table list, handle table of the process csrss.exe and PspCidTable. The first one is a linked list, and the DKOM behavior of FUTo over this kernel structure was captured and displayed in the tag trace graph too (seetag ID 36, 37 and 38). The last two kernel structures are implemented as three-dimensional arrays, which is not supported by the current version of Rkprofiler. So, the tag trace graph does not include the modification of these two kernel structures.

Combining Rkprofiler's output with other reports, we discovered other interesting behavior of FUTo. First, FUTo employed an IOCTL mechanism to pass control commands from user space to kernel space. During the driver initialization, a device \\Device\\msdirectx was created by calling the kernel function IoCreateDevice. Then a dispatch function (data type FUNCT_0049_095B_Majorfunction and tag ID 7) was registered to the driver object (with tag ID 1) that was assigned to msdirectx by the Windows kernel. This dispatch function was invoked by the kernel I/O manager to process I/O requests issued by the fu.exe processes. By checking the parameters of this dispatch function, we found that the I/O control codes for process and driver concealment tasks are 0x2a7b2008 and 0x2a7b2020. Second, the kernel string function strncmp was called 373 times by one msdirectx function, implying a brute-force searching operation. The first parameter of this function was constant string ``System'' and the second parameter was 6 bytes of data within the process descriptor of the process System (with tag ID 2). Beginning with the address of the process descriptor, the address of the second parameter was increased by one byte each time this string function was called. The purpose of the search was to find the offset of the

process name in the EPROCESS structure. This was confirmed by manually checking the FUTo source. It seems that the definition of EPROCESS structure has changed over the Windows versions and the brute-force searching allows FUTo to work with different Windows versions.

4.8.2 TCPIRPHOOK

Inserting hooks into the kernel to tamper with the kernel control-flow path is one major technique that attackers apply to rootkit attacks. A hooked function can intercept and manipulate kernel data to serve its malicious aims. TCPIRPHOOK is one such rootkit and it intends to hide the TCP connections from local users. Specifically, this rookit exploits the dispatch function table of the TCP/IP driver object (associated with driver TCPIP.sys) and substitutes a dispatch function with its hook. The hooked function registers another hook to the I/O request packets (IRP) such that the second hook can intercept and modify the query results for network connections. We downloaded the rootkit package from [66] which also included one driver file, irphook.sys. The rootkit was implemented to conceal all http connections (with destination port 80). Before installing the rootkit, we opened Internet Explorer to visit a few websites, and then ran the netstat utility to display the corresponding http connections. We loaded the irphook.sys to the kernel and used netstat to verify that all https connections were gone. In the end, we unloaded the irphook.sys. The test took less than 3 minutes.

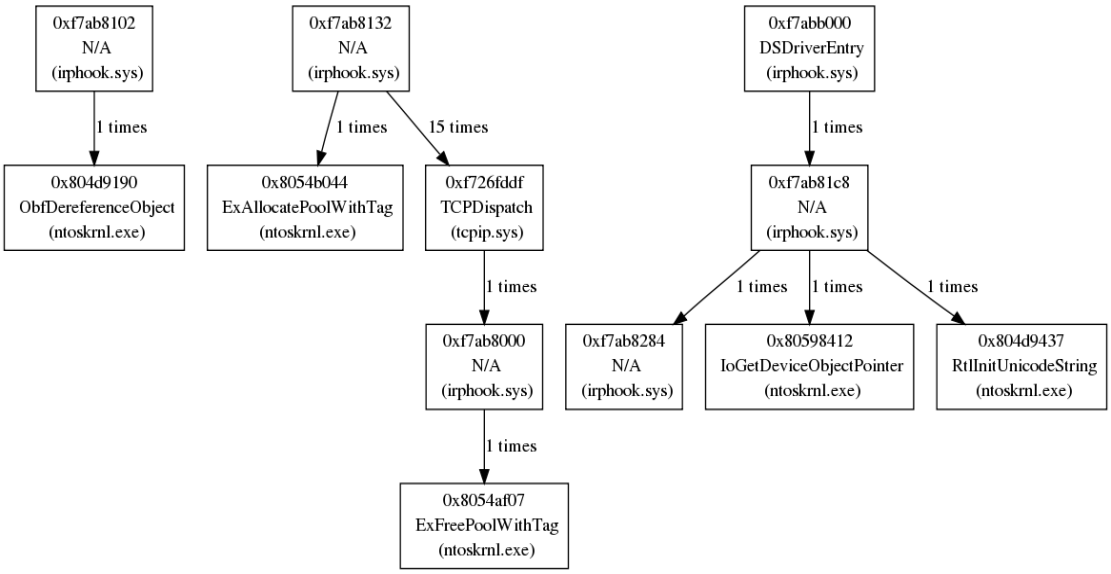


Fig. 6. TCPIRP call graph

The call graph of TCPIRPHOOK is shown in Figure 6. Function 0xf7ab8132 (irphook.sys) was the first hook that was inserted into the 14th entry (IRP_MJ_DEVICE_ CONTROL) of the dispatch function table in the driver TCPIP.sys. The replaced dispatch function was TCPDispatch (address 0xf726fddf) owned by driver TCPIP.sys. The first hook invoked TCPDispatch 15 times in the call graph. In fact, it is common for rootkits to call the original function in a hook, which reduces the coding complexity of the hook. Function 0xfa7b8000 (irphook.sys) was the second hook that was responsible for modifying the query results for network connections. Although the second hook seems to be called by TCPDispatch in the call graph, the actual direct caller of the second hook was IopfCompleteRequest (ntoskrnl.exe). This is because Rkprofiler did not track the benign kernel code and had no knowledge of their call stacks. On the other hand, even the indirect caller-callee relation between TCPDisptch and the second hook can imply that the network connection query caused synchronous IRP processing and completion in the kernel, which is comparable to asynchronous IRP processing and completion. But this information cannot be inferred by simply looking at the call graph of IDA-pro,

because IDA-pro cannot statically determine the symbol of function TCPDispatch and the calling path from the first hook to the second hook in Figure 3 is not presented in the call graph of IDA-pro.

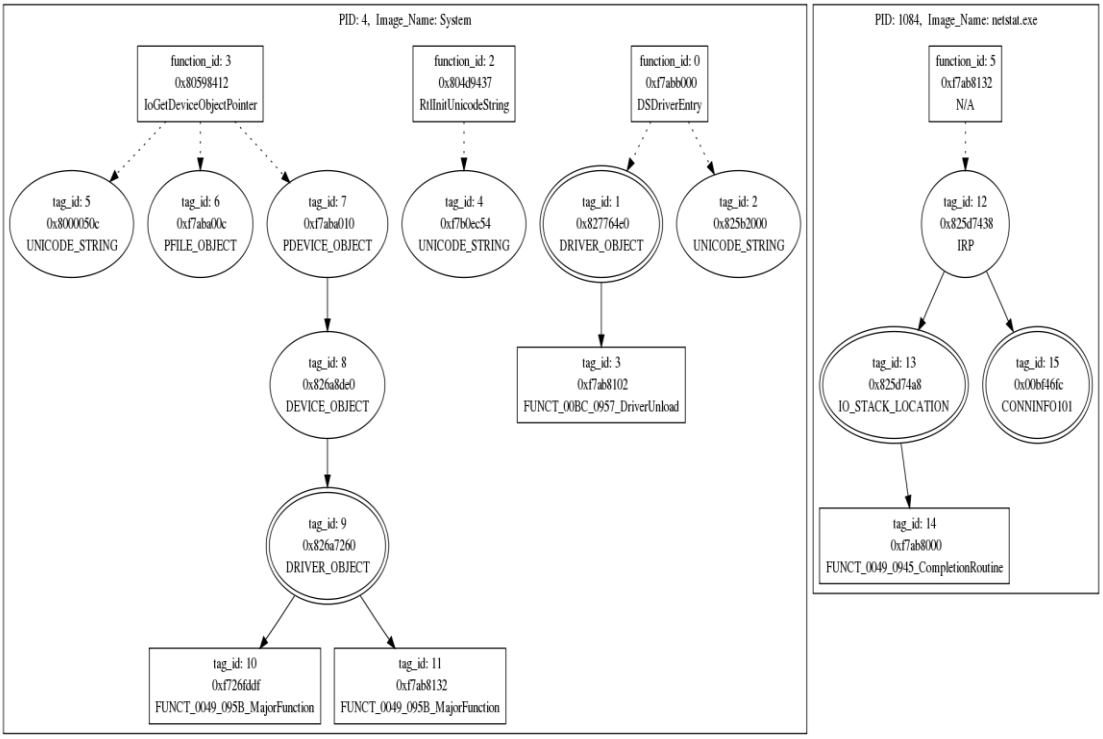


Fig. 7. TCPIRP tag trace graph

Figure 7 is the tag trace graph of TCPIRPHOOK. Two hooking activities are illustrated in this graph. The first hook was installed at the driver loading stage. To hook the dispatch function table of the driver TCPIP.sys, TCPIRPHOOK first calls the kernel function IoGetDeviceObjectPointer with the device name\\Device\\Tcp to get the pointer (with tag ID 7) to the device object (withtag ID 8) owned by driver TCPIP.sys. Then, the device object was visited to get the address of the driver object (withtag ID 9) owned by driver TCPIP.sys. Last, TCPIRPHOOK carried out the hooking by accessing the 14th entry of the dispatch function table in the driver object: reading the address of the original dispatch function (with tag ID 10) and storing it to a global variable; writing the address of the second hook (withtag ID 11) to the table

entry. The second hook was dynamically installed in the context of process netstat.exe. When netstat.exe was executed to query TCP connection status, the Windows kernel I/O manager created an IRP (withtag ID 12) for the netstat.exe process. This IRP was passed to the first hook function_id 5 and tag ID 11) of TCPIRPHOOK. The first hook obtained the IO_STACK_LOCATION object (with tag ID 13) from this IRP and wrote the address of the second hook (withtag ID 14) to the data member CompletionRoutine of the IO_STACK_LOCATION object. Thus, being one IRP completion function, the second hook would be called by the Windows kernel to process the I/O return data for this IRP. Last, the tag trace graph also captures the manipulation of the I/O return data. The buffer of the I/O return data was pointed to by the data member UserBuffer of IRP and it was an array of structure CONNINF101 (withtag ID 15). The size of the buffer was stored in the data member IoStatus.Information of the IRP. Clearly, the tag ID 15 was modified in the tag trace graph. By examining the tag trace table, we found that the statuses of all http connections in the buffer were changed from 5 to 0.

4.8.3 Shadow Walker

Shadow Walker is a virtual memory subversion rootkit. It makes other kernel malware invisible to detectors by controlling virtual memory mappings. For read/wire virtual memory access, Shadow Walker presents a benign page of memory; for execute access on the same virtual address, it will execute the hidden code. Concretely, it replaces the page fault handler in the IDT with a hook and marks the hidden pages in the page table “not present”. When a hidden page is visited by CPU, a page fault exception is generated and passed to the hook of Shadow Walker. The hook compares the faulting address in CR2 with the instruction pointer in ESP: if they are equal, then it is a code access; otherwise, it is a data access. Then, the hook returns a

mapping to either the actual rootkit code or random data accordingly. We downloaded the Shadow Walker from [66]. The download package contained two drivers: msdirectx.sys and mmHook.sys. The first one is Futo rootkit that Shadow Walker attempts to hide. The second one is the Shadow Walker driver. Because we just wanted to test the behaviors of Shadow Walker, Futo rootkit was treated as benign in the experiment. We first loaded msdirectx.sys and then started the rkprofiler's monitoring function. Last, we loaded the mmHook.sys. No particular user-space application was executed in this experiment.

In the Rkprofiler reports, we found that some malicious code was executed periodically in the interrupt contexts. The IDT vector number was 14, indicating that the page fault handler was hacked. In addition, the following three system instructions were executed multiple times: SIDT, MVCR2 and INVLPG. SIDT was used by Shadow Walker to read the address of IDT and substituted the original page falut handler with the hook in IDT; MVCR2 allowed the hook to read the CPU control register 2 and get the faulting address upon a page fault exception; Shadow Walker used the INVLPG to flush a particular TLB entry and to synchronize the data TLB, code TLB and page tables. As no data type definitions for page table and IDT were supplied to Rkprofiler, we do not expect the Rkprofiler can detect IDT hooking and page table manipulation in the experiment.

4.8.4 Rustoc.B

Rustock.B is a notorious backdoor rootkit that hides malicious activities on a compromised machine. The distinguished feature of this rootkit is the usage of multi-layered code packing, which makes static analysis cumbersome [57]. Unlike the other two rookits described above, we did not have access to the source code of this rootkit.

However, several analysis results on this rootkit published on the Internet helped us understand some behaviors of this rootkit. We downloaded Rustock.B from [67] as one executable. During the test, we just double-clicked the binary and waited until the size of the Rkprofiler log stop being populated. The test lasted about five minutes.

Table 9. External functions and registry keys manipulated by Rustock.B

External Functions	ExAllocatePoolWithTag, ExFreePoolWithTag, ExInitializeNPagedLookasideList, IoAllocateMdl, IoGetCurrentProcess, IoGetDeviceObjectPointer, IoGetRelatedDeviceObject, KeClearEvent, KeDelayExecutionThread, KeEnterCriticalRegion, KeInitializeApc, KeInitializeEvent, KeInitializeMutex, KeInitializeSpinLock, KeInsertQueueApc, KeLeaveCriticalRegion, KeWaitForSingleObject, MmBuildMdlForNonPagedPool, MmMapLockedPages, MmProbeAndLockPages, NtSetInformationProcess, ObfDereferenceObject, ObReferenceObjectByHandle, ProbeForRead, PsCreateSystemThread, PsLookupProcessByProcessId, PsLookupThreadByThreadId, RtlInitUnicodeString, _stricmp, _strnicmp, swprintf, wcschr, wcscpy, _wcsicmp, _wcslwr, wcsncpy, _wcsnicmp, wcstombs, ZwClose, ZwCreateEvent, ZwCreateFile, ZwDeleteKey, ZwEnumerateKey, ZwOpenKey, ZwQueryInformationFile, ZwQueryInformationProcess, ZwQuerySystemInformation, ZwReadFile
Registry Keys	HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\pe386 HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\ Root\LEGACY_pe386

A malicious driver named system32:lzx32:sys was detected by Rkprofiler. 90857 calls and 2936 tags were captured in the test. The driver contained self-modifiying code and we found many RET instructions that did not have corresponding CALL instructions at code unpacking stages. This is because unpacking routines executed JMP instructions to transfer the controls to the intermediate or unpacked code. In addition, the driver modified the dedicated register IA32_SYSENTER_EIP through WRMSR and RDMSR instructions to hijack the Windows System Service Descriptor Table (SSDT). One hook was added to the dispatch function table of driver Ntfs.sys to replace the original IRP_MJ_CREATE dispatch function. This is similar to what TCPIRPHOOK does. We compared the report generated by Rkprofiler with others on

the Internet and they matched each other well. Table 9 lists the external functions and registry keys that were called and created by Rustock.B.

4.9 Discussion

In addition to the incomplete kernel symbols provided by Microsoft, the current implementation of Rkprofiler suffers several other limitations that could be exploited by attackers to evade the inspection. First, attackers may compromise the kernel without running any malicious kernel code, e.g., directly modifying kernel data objects from user space or launching return-to-lib attacks without the use of any function calls and malicious kernel code [68] [74]. Rkprofiler is not able to detect and profile such attacks. Instead, other defense approaches like control flow integrity enforcement [69] could be adopted to address them. Second, the instruction pair CALL/RET is used as the sole indicator of function call and return events. Attackers can obfuscate these function activities to escape the monitoring. For example, JMP/JMP, CALL/JMP and JMP/RET can be employed to implement the function call and return events. Moreover, instead of jumping to a target instruction (either the first instruction of a callee function or the returned instruction of a caller function), an attacker could craft the code to jump to one of its neighbor instructions, while preserving the software logic intact. Defending against such attacks is part of our future work. Third, an attacker may deter the AMT method by accessing dynamic objects in unconventional ways. For example, a rootkit can scan the stack of a benign kernel function to get the pointer to a desired kernel object. These attacks are very challenging, because building an accurate and up-to-date symbol table for all kernel objects is impractical. Last, malware may have the capability of detecting virtual machine environments and change their behavior accordingly. Min Gyung Kang [75] presents several countermeasures to address this problem.

CHAPTER 5

CONCLUSIONS AND RECOMMENDATIONS

Although kernel malware is a nightmare to security software running on regular physical machines, VM-based computing systems provide vital advantage to security software in the war against kernel malware: higher system privilege. By taking advantage of this, we designed and implemented two VM-based security systems in this research work to counter the kernel malware who run inside guest operating systems.

DARK is a rootkit prevention system that applies on-demand emulation technique to dynamically monitor, detect and block suspicious LKMs. We created a group of state-of-art security policies for DARK. The evaluation results demonstrated that DARK was able to effectively defeat the Linux kernel rootkits that are available to us, while keeping the reasonable system performance. Rkprofiler is a rookit analysis system that thoroughly monitors and reports the behaviors of Windows kernel rootkits. In particular, aggressive memory tagging (AMT) is proposed to resolve the symbols of dynamic kernel objects. We used the Rkprofiler to analyze a number of real-world Windows rootkits, showing that substantial rootkit behaviors were revealed by Rkprofiler.

However, as we pointed out in the Chapter 3 and 4, both DARK and Rkprofiler have limitations, so more research efforts are needed to address them down to the road. Here is list of future work:

1. Find a way to help DARK user automatically identify the suspicious LKMs and trigger DARK's monitoring. Driver signing and static driver analysis are two options for this purpose.

2. Further reduce the performance overhead caused by emulation and LKM monitoring in DARK, e.g., identifying the benign execution paths of a suspicious module.
3. Automate the security policy generation in DARK by applying data mining technique to automatically extract rules from the training set.
4. Explore new mechanism to accurately identify ambiguous data types (UNION, generic pointer and dynamic array), which could improve the symbol coverage of Rkprofiler significantly.
5. Design new method to reliably detect function call and return events in Rkprofiler, increasing the resistance to evasion techniques possibly used by rookits, e.g., JMP/JMP pair.
6. Find proper technique to explore and profile the multiple execution paths of kernel malware, which is more challenging than that of user-space malware presented in [70].

REFERENCES

[1] Driver Signing Requirements for Windows, http://www.microsoft.com/whdc/winlogo/drvsign/drvsign.mspx, 2009

[2] S. Field, "x64 Driver Signing Update," http://blogs.msdn.com/windowsvistasecurity/archive/2007/08/03/x64-driver-signing-update.aspx, Aug 2007.

[3] J Rutkowska, A Tereshkin, "IsGameOver(), anyone?," BlackHat USA 2007.

[4] H. Yin, Z. Liang, and D.Song, "Hookfinder: Identifying and understanding malware hooking behaviors," Proceeding of the Annual Network and distributed System Security Symposium (NDSS), 2008.

[5] J. Wilhelm and T. Chiueh, "A Forced Sampled Execution Approach to Kernel Rootkit Identification," Proceedings of International Symposium on Recent Advances in Intrusion Detection (RAID), 2007.

[6] A. Lanzi, M. Sharif, W. Lee, "K-Tracer: A System for Extracting Kernel Malware Behavior," Proceeding of the Annual Network and distributed System Security Symposium (NDSS), 2009.

[7] Wikipedia. "Rootkit," 2009: http://en.wikipedia.org/wiki/Rootkit.

[8] Uninformed, "FUTo," January 2006: http://uninformed.org/ v=3&a=7&t=sumry.

[9] J. Rutkowska, "Introducing Stealth Malware Taxonomy," 2006: http://www.invisiblethings.org/papers.html.

[10] Blue Pill Project, http://bluepillproject/, 2009.

[11] Intel Virtualization Technology, http://www.intel.com/technology/virtualization/, 2009.

[12] AMD Virtualization, http://www.amd.com/us/products/technologies/virtualization/Pages/virtualization.aspx, 2009.

[13] Free BSD jails, http://www.freebsd.org/, 2009.

[14] OpenVZ Project, http://wiki.openvz.org/Main_Page, 2009.

[15] J. Sugerman, G. Venkitachalam and B. Lim, "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor," Proceedings of the USENIX Annual Technical Conference (USENIX), 2001.

[16] P. Barham, B. Dragovic, K. Fraser, S. Hand, T.L.Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), 2003.

[17] Key Trends: Virtualization security. http://www.nemertes.com/key_trends/key_trends_virtualizations_ecurity, 2009.

[18] T. Garfinkel and M. Rosenblum., "A Virtual Machine Introspection Based Architecture for Intrusion Detection," Proceedings of the Symposium on Network and Distributed System Security (NDSS), 2003.

[19] X. Jiang, X. Wang, D. Xu, "Stealthy Malware Detection Through VMM-Based "Out-of-the-Box" Semantic View Recontruction," Proceedings of the ACM Conference on Computer and Communications Security (CCS), 2007.

[20] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, D. Boneh, "Terra: A Virtual Machine-Based Platform for Trusted Computing," Proceedings of the ACM Symposium on Operating systems Princeiples (SOSP), 2003.

[21] X. Chen et atl. "Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems," Proceedings of the Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2008.

[22] G. W. Dunlap et al., "ReVirt: Enabling intrusion analysis through virtual-machine logging and replay," Proceedings of the ACM SIGOPS Operating Systems Review, 2002.

[23] J. Chow, T. Garfinkel, and P. M. Chen, "Decoupling Dynamic Program Analysis from Execution in virtual Environments," Proceedings of the USENIX Annual Technical Conference (USENIX), 2008.

[24] Microsoft. "Windows Kernel Patch Protection," 2008: http://www.microsoft.com/whdc/driver/kernel/64bitpatching.mspx.

[25] Uninformed, "PatchGuard Reloaded. A Brief Analysis of PatchGuard Version 3," September 2007: http://uninformed.org/index.cgi?v=8&a=5.

[26] N. L. Petroni, T. Fraser, A. Walters, and W. A. Arbaugh, "An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data," Proceedings of the USENIX Security Symposium, 2006.

[27] N. L. Petroni, M. Hicks, "Automated Detection of Persistent Kernel Control-Flow Attacks," Proceedings of the ACM Conference on Computer and Communications Security (CCS), 2007.

[28] R. Riley, X. Jiang and D. Xu, "Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing," Proceedings of International Symposium on Recent Advances in Intrusion Detection (RAID), 2008.

[29] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "VMM-based hidden process detection and identification using Lycosid," Proceedings of the 4th International Conference on Virtual Execution Environments (VEE), 2008.

[30] L. Litty, H. A.Lagar-Cavilla, D. Lie, "Hypervisor Support for Identifying Covertly Executing Binaries," Proceedings of the USENIX Security Symposium, 2008.

[31] A. Baliga, V. Ganapathy and L. Iftode, "Automatic Inference and Enforcement of Kernel Data Structure Invariants," Proceedings of the Annual Computer Security Applications Conference (ACSAC), 2008.

[32] X. Zhang, L. van Doorn T. jaeger, R. Perez, and R. Sailer, "Secure Coprocessor-based Intrusion Detection," Proceedings of the ACM SIGOPS European Workshop, 2002.

[33] N. L. Petroni, T. Fraser, J. Molinz, and W. A. Arbaugh, "Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor," Proceedings of the USENIX Security Symposium, 2004.

[34] A. Seshadri, M. Luk, E. shi, A. Perrig, L. vanDoorn, and P. Khosla, "Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems," Proceedings of the ACM Symposium on Operating systems Princeiples (SOSP), 2005.

[35] G. Kim and E. Spafford, "The Design and Implementation of Tripwire: A File System Integrity Checker," Proceedings of the ACM Conference on Computer and Communications Security (CCS), 1993.

[36] A. Baliga, P. Kamat and L. Iftode, "Lurking in the Shadows: Identifying Systemic Threats to Kernel Data," Proceedings of IEEE Symposium on Security and Privacy, 2007.

[37] Y.M. Wang, D. Beck, B. Vo R. Roussev, C. Verbowski, "Detecting Stealth Software with Strider GhostBuster," Proceeding of International Conference on Denpendable Network Systems (DSN), 2005.

[38] A. Seshadri, M. Luk, N. Qu and A. Perrig, "SecVisor: A Tiny Hypervisor to Guarantee Lifetime Kernel Code Integrity for Commodity OSes," Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), 2007.

[39] B C. Kruegel, W. Robertson and G. Vigna, "Detecting Kernel-Level Rootkits Through Binary Analysis," Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC), 2004.

[40] J. Wilhelm and T. Chiueh, "A Forced Sampled Execution Approach to Kernel Rootkit Identification," Proceedings of International Symposium on Recent Advances in Intrusion Detection (RAID), 2007.

[41] R. Riley, X. Jiang, D. Xu, "Multi-Aspect Profiling of Kernel Rootkit Behavior," Proceedings of the ACM SIGOPS European Conference on Computer Systems (EuroSys), 2009

[42] N.Jethercote and J. Seward, "Valgrind: A Program Supervision Framework," Proceedings of the Third Workshop on Runtime Verification, 2003.

[43] K. Scott and J. Davidson, "Safe Virtual Execution Using Sofware Dynamic Translation," Proceedings of the Annual Computer Security Applications Conference (ACSAC), 2002.

[44] V. Kiriansky, D. Bruening, and S. P. Amarasinghe, "Secure Execution via Program Shepherding," Proceedings of the USENIX Security Symposium, 2003.

[45] A. Ho, M. Fetterman, C. Clark, A. Warfield, S. Hand, "Practical Taint-Based Protection using Demand Emulation," Proceedings of the ACM SIGOPS European Conference on Computer Systems (EuroSys), 2006.

[46] F. Bellard, "Qemu, a Fast and Portable Dynamic Translator," Proceedings of the USENIX Annual Technical Conference (USENIX), 2005.

[47] G. Hoglund, J. Butler, "Rootkits: Subverting the Windows Kernel," Addison-Wesley Professional, August 2005.

[48] Francis M. D, Ellick M. C, Jeffery C. C and Roy, C, "Cloaker: Hardware Supported Rootkit Concealment," Proceedings of IEEE Symposium on Security and Privacy, 2008.

[49] J. Heasman, "Implementing and Detecting a PCI Rootkit," Technical report, next Generation Security Software Ltd, November 2006.

[50] J. Heasman. Implementing and Detecing an ACPI BIOS Rootkit. In Black Hat Europe, Amsterdam, March 2006.

[51] S. T. King, et atl, "Subvert: Implementing Malware with Virtual Machines," Proceedings of the IEEE symposium on Security and Privacy, 2006.

[52] Scythale. Hacking deeper in the system. http://www.phrack.com/issues.html?issue=64&id=12#article

[53] bonnie. http://www.textuality.com/bonnie/, 2009.

[54] nbench. http://www.tux.org/~mayer/linux/bmark.html, 2009.

[55] Iperf. http://dast.nlanr.net/Projects/Iperf/, 2009.

[56] P. Ferrie, "Attacks on virtual machine emulators," Symantec Security Response. 2006.

[57] K. Chiang and L. Lloyd, “A case Study of the Rustock Rootkit and Spam Bot,” Proceedings of the First Workshop on Hot Topics in Understanding Botnets (HotBots), 2007.

[58] Dr Web, http://info.drweb.com/show/3342/en, 2009.

[59] http://www.cbsnews.com/stories/2009/04/09/tech/cnettechnews/main4931360.shtml, 2009.

[60] http://geeg.info/blog4.php/2009/04/the-conficker-worm-awakens, 2009.

[61] Anubis Project, http://anubis.iseclab.org/?action=home, 2009.

[62] D. Song, D. Brumley et al., “BitBlaze: A new Approach to Computer Security via Binary Analysis,” Proceedings of the International Conference on Information Systems Security, 2008.

[63] Microsoft Symbol Server, http://msdl.microsoft.com/download/symbols, 2009.

[64] GraphViz Project, http://www.graphviz.org/, 2009.

[65] MSDN, “Statck Allocation,” 2009, available: http://msdn.microsoft.com/en-us/library/ms794596.aspx.

[66] www.rootkit.com, 2009.

[67] http://www.offensivecomputing.net/, 2009.

[68] H.Shacham, “The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls,” Proceedings of the ACM Conference on Computer and Communications Security (CCS), 2007.

[69] M. Abadi, M. Budiu, U. Erlingsson, and j. Ligatti, “Control-flow integrity: Principles, implementations, and applications,” Proceedings of the ACM Conference on Computer and Communications Security (CCS), 2005.

[70] A. Moser, C. Kruegel, and E. Kirda, “Exploring multiple execution paths for malware analysis,” Proceedings of the IEEE Symposium on Security and Privacy, 2007.

[71] B. D. Gavitt, A. Srivastava, P. Traynor, J. Giffin, “Robust Signatures for Kernel Data Structures,” Proceedings of the ACM Conference on Computer and Communications Security (CCS), 2009.

[72] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, X. Jiang, “Mapping Kernel Objects to Enable Systematic Integrity Checking,” Proceedings of the ACM Conference on Computer and Communications Security (CCS), 2009.

[73] Z. Wang, X. Jiang, W. Cui, P. Ning, "Countering Kernel Rootkits with Lightweight Hook protection," Proceedings of the ACM Conference on Computer and Communications Security (CCS), 2009.

[74] R. Hunk, T. Holz, F. C. Freiling, "Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms," Proceedings of the USENIX Security Symposium, 2009.

[75] M. G. Kang, H. Yin, S. Hanna, "Emulating Emulation-Resistant Malware," Proceedings of the 2nd Workshop on Virtual Machine Security (VMSec), 2009.

[76] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," Science of Computer Programming, 69(1-3), 2007.

[77] J. Rutkowska, "Beyond the CPU: Defeating Hardware Based RAM Acquision," Black Hat, 2007.

[78] G. J. Popek and R. P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," Communications of the ACM 17(7): 412-421, 1974.

[79] J. S. Robin and C. E. Irvine, "Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor," Proceedings of the USENIX Security Symposium, 2000.

[80] VMware, "Performance Evaluation of Intel EPU Hardware Assist," 2009, available: http://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf.

[81] Boch Project, http://bochs.sourceforge.net/, 2009.

[82] Qemu and Kqemu Project, http://www.qemu.org/, 2009.

[83] J. E. Smith and R. Nair, "Virtual Machine: Versatile Platforms for Systems and Processes," Morgan Kaufmann, 2005.

PUBLICATIONS

[1] Chaoting Xuan, John Copeland, and Raheem Beyah, “Toward Revealing Kernel Malware Behavior in Virtual Execution Environments,” Proceedings of International Symposium on Recent Advances in Intrusion Detection (RAID), Sep 2009.

[2] Chaoting Xuan, John Copeland, and Raheem Beyah, “Shepherding Loadable Kernel Module through On-demand Emulation,” Proceedings of SIG SIDAR Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), July 2009.

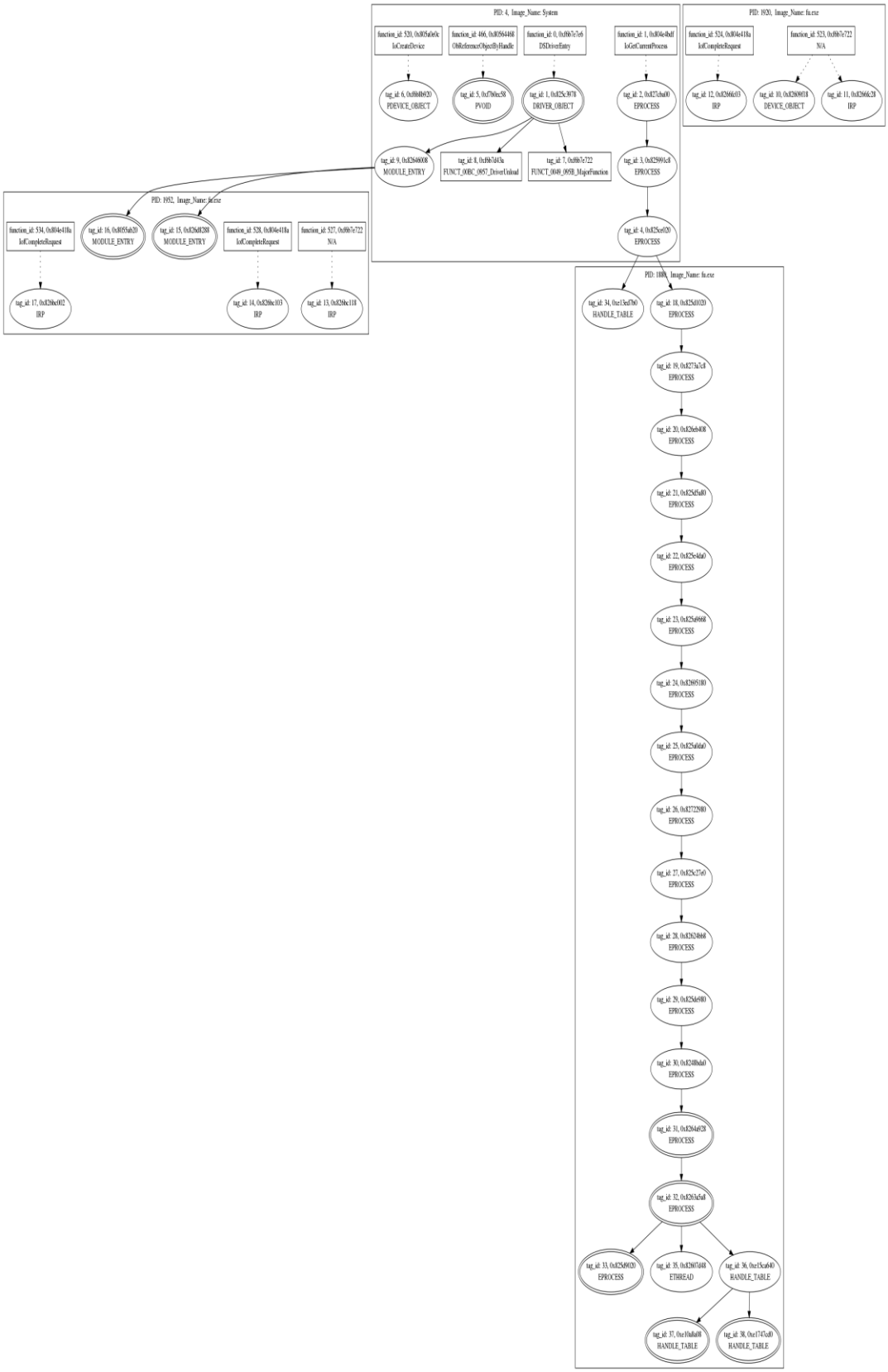
[3] Chaoting Xuan, Mustaque Ahamad. “SID: PKI-enabled Identity Management System,” Proceedings of IEEE Conference on E-commerce Technology (CEC), July 2004.

[4] Jeffrey King, Andre dos Santos, and Chaoting Xuan, "KHAP: Using Keyed Hard AI Problems to Secure Human Interfaces," Scientia, vol. 15, n. 01, January/June 2004, pp. 50-59.

APPENDIX A – Imbench TEST RESULT

```
Processor, Processes - times in microseconds - smaller is better
----------------------------------------------------------------------------
--
System             OS  Mhz null null     open slct sig  sig  fork exec sh
                         call  I/O stat clos TCP  inst hndl proc proc proc
--------- ------------- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- --
--
VMM      Linux 2.4.18- 2184 7.39 7.61 195. 198. 67.1 7.67 106. 4807 13.K 34.K
DARK-CS   Linux 2.4.18- 2184 7.24 7.63 189. 219. 67.3 7.67 104. 4925 13.K
35.K
DARK     Linux 2.4.18- 2184 7.34 7.60 190. 198. 66.2 7.87 107. 4943 13.K
35.K

Context switching - times in microseconds - smaller is better
-------------------------------------------------------------------------
System             OS  2p/0K 2p/16K 2p/64K 8p/16K 8p/64K 16p/16K 16p/64K
                         ctxsw  ctxsw  ctxsw ctxsw  ctxsw   ctxsw   ctxsw
--------- ------------- ------ ------ ------ ------ ------ ------- -------
VMM      Linux 2.4.18-  140.8  164.1        189.0         201.8
DARK-CS   Linux 2.4.18-  158.4  183.8        205.0         224.2
DARK     Linux 2.4.18-  162.6  192.8        209.3         228.5

*Local* Communication latencies in microseconds - smaller is better
---------------------------------------------------------------------
System             OS 2p/0K  Pipe AF    UDP  RPC/   TCP  RPC/ TCP
                        ctxsw       UNIX         UDP         TCP conn
--------- ------------- ----- ----- ---- ----- ----- ----- ----- ----
VMM      Linux 2.4.18- 140.8 278.9 863. 891.5 1666. 1721. 2542. 3958
DARK-CS   Linux 2.4.18- 158.4 273.8 866. 9810. 1985. 1981. 2643  3594
DARK     Linux 2.4.18- 162.6 276.4 849. 9396. 2001. 1920. 2745  3676

File & VM system latencies in microseconds - smaller is better
-------------------------------------------------------------------------------
---
System             OS   0K File     10K File    Mmap    Prot   Page   100fd
                         Create Delete Create Delete Latency Fault  Fault  selct
--------- ------------- ------ ------ ------ ------ ------- ----- ------- --
---
VMM      Linux 2.4.18-  529.9  533.3 1447.2 1081.1   669.0  15.4    20.0  64.3
DARK-CS   Linux 2.4.18-  510.7  552.8 1697.8  838.9   772.0  13.9    24.6  63.7
DARK     Linux 2.4.18-  530.1  530.9 1579.0 1203.8   715.0  15.7    25.7  64.7

*Local* Communication bandwidths in MB/s - bigger is better
----------------------------------------------------------------------------
-
System            OS  Pipe AF    TCP  File   Mmap  Bcopy  Bcopy  Mem   Mem
                            UNIX      reread reread (libc) (hand) read write
--------- ------------- ---- ---- ---- ------ ------ ------ ------ ---- ----
-
VMM      Linux 2.4.18- 18.0 9.57 5.05 1506.2 1879.1  510.7  484.1 1262 950.0
DARK-CS   Linux 2.4.18- 17.8 9.21 4.63 1291.5 1423.2  437.6  418.8 1195 904.7
DARK     Linux 2.4.18- 17.7 8.91 4.31 1204.8 1980.2  835.9  474.9 1217 945.9
```

APPENDIX B – TAG TRACE GRAPH OF FUTo



VITA

Chaoting Xuan

Born in Zhengzhou, China, Chaoting Xuan is one of the four children of Jiarang Xuan and Xiuying Zhu, both of whom are school teachers. In July of 1997, Chaoting graduated from Shanghai Jiao Tong University, China, with B.S. degree in Material Science and Engineering. He came to United States in 2000 and started graduate studies here. Chaoting respectively received M.S. degree in Electrical and Computer Engineering from North Carolina State University, Raleigh, North Carolina in 2002, and M.S. degree in Computer Science from Georgia Institute of Technology, Atlanta, Georgia in 2004. He was enrolled in the PhD program of the department of Electrical and Computer Engineering in 2006. Starting from 2004, Chaoting works for Trusted Network Technologies (acquired by Liquidware Labs) as software engineer until today. He married to Hua Tu in 2000 and they have two children: Henry and Grace.